In Proceedings Recent Advances in PVM and MPI, M. Bubak, J. Dongarra and J. Wasniewski (eds.), LNCS 1332, 1997, pp.242-249

Message-Passing Program Development by Ensemble

J.Y. Cotronis
Dept. of Informatics, Univ. of Athens, Panepistimiopolis, 157 71 Athens, GREECE.
tel.: +30 1 7230172 fax: +30 1 7219 561 e-mail: cotronis@di.uoa.gr

Abstract
We present Ensemble, a message-passing implementation methodology, applied to PVM. Ensemble overcomes problems and complexities in developing applications in message-passing environments (MPE). Applications are specified by scripts, which represent Process Communication Graphs (PCG) annotated with information specific for an MPE, and by reusable executable components. A loader program interprets the scripts and composes applications from reusable components.

1 Motivation

The design and implementation of parallel message passing applications (MPA) have been recognized as demanding tasks. PVM [5] and MPI [6] Message Passing Environments (MPE) have improved the situation, as they permit the implementation of applications independently of the underlying architecture, but the step from application design to implementation remains, in general, a demanding task. The implementation does not only require the programming of sequential code solving a problem or providing a service, but also requires the explicit programming of process management, e.g. process creation, process topologies and their mapping onto the architecture. As MPEs differ significantly in the way they manage processes, message passing program implementation presents a number of difficulties. MPEs impose on application implementation their own structure to such a degree, that implementations of the same application design (even a simple ring topology) on different MPEs, using the same language for the sequential computations, look very different. The implementation difficulty does not depend only on the complexity of the design, but also on the particular MPE used. Some applications are more easily implemented on some MPEs than on others. For example, a general tree topology is easy to implement on PVM, but difficult on MPI. The difficulty arises from the process management model each MPE uses. Often, the original application design is obscured in the implementation by the required process management programming.

Consequently, implementation requires special techniques and skills, different for each MPE. There are no methodologies for implementing applications on PVM or MPI. Programmers have to implement relying on their experience, quite often in an ad hoc way, developing programs that are difficult to debug, extend and modify. Important aspects of parallel programming, such as scalability, reusability are often neglected. Scalability is programmed only when the design has some regularity, as expressed by some function; the degree of difficulty for programming scalability also depends on the MPE used. Reusability of executables is limited as process management (creation, topology, mapping, etc.) is encoded in them; processes are programmed to operate in a fixed topology of known processes.

We have developed a message passing program implementation methodology, called Ensemble, for overcoming the above problems. The original design is maintained in the implementation, programs implementing the same designs in different MPEs look similar, are easily maintainable, scalable, reusable and portable to other MPEs.

Ensemble comprises three aspects: 1. The annotated Process Communication Graphs (PCG). PCGs have been extensively used in modeling, dynamic analysis and simulation [10,11], in mapping techniques [1,8], etc. and depict the topology of the processes involved in an application. We *annotate* nodes and arcs of PCGs with information required for the process creation and communication for specific MPEs. 2. The reusable program components. Their executables are message passing *library components*, as they do no involve any process management and do not rely on any specific processes or topologies for communication; MPEs require different techniques for developing such components. 3. The Loader. *Interprets* the annotated PCGs and *composes* applications; the Loader visits the nodes of the annotated PCGs, spawns processes, which are instantiations of the reusable program components, and establishes process topologies. Each MPE requires its own Loader, but once developed it may be used to compose applications within an MPE.
In section 2, we outline the Ensemble methodology, its techniques and tools for PVM; in 3, we demonstrate the reusability of components and in 4, we present our conclusions.

2 Ensemble for PVM

PVM allows for the most general form of parallel computation, as programs may exhibit arbitrary communication dependencies [6]. As with all programming environments, though, there are program categories suited to PVM, making them easy to implement and others not well suited. Programs forming, in general, tree process communication dependencies (including master-slave, as a one level tree, and SPMD, as a forest of roots), where each process communicates only with its parent and its children processes, are well suited to PVM. Processes are spawned by parent processes as children-processes. Communication between parent and children processes is established by using their respective tids and by tagging the messages they exchange by the same integer identifier (tagid). Developing programs forming general graph process topologies is a complex task in PVM. Establishing graph process topologies, even a simple ring, requires the explicit programming of (a) creating processes according to the parent-child model, and (b) establishing process communication, by obtaining the tids of the processes with which they need to communicate; processes get their children's tids and they may easily obtain their parent's tid, but they must also obtain the rest of the tids. This explicit programming is an overhead effort, it burdens the original application design and it limits the reusability, scalability and portability of programs.
We present Ensemble by implementing an application, the Distribution of Maximum: *terminal* processes hold an integer value and require the maximum of all values. We develop the following design: each terminal process sends its value to an associated *relay* process and (eventually) receives from it the global maximum (a client-server model). Relay processes (as servers) find the local maximum of their associated terminals, they exchange their local maxima, they find the global maximum and send it to their associated terminal processes (as clients).

2.1 The annotated PCG

The annotated PCG of the application is specified by a script, which has three main parts. The first, headed by PCG, specifies the PCG of the application. The second, headed by PARALLEL SYSTEM, specifies the annotation of the PCG for a specific parallel environment (in this case PVM3). The third, headed by SEQUENTIAL COMPONENTS, specifies the further annotation of the PCG with information for the sequential components. Each of the three parts consists of sections. The script and its associated PCG and part of its annotation is shown in fig. 1.

The two components, terminal and relay, and their communication dependency types are specified in the Components section of the PCG part. Terminal processes have one type of communication dependency, that with their associated relay processes, which we call *Server* type. Relay processes have two types of communication dependencies, one with their terminal processes, which we call *Client* type, and one with the relay processes, which we call *Prop* (propagation) type.

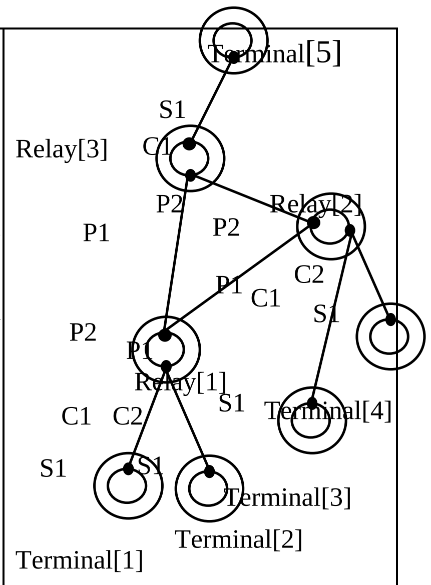
The actual number of processes and the number of their communication dependencies of each type are described in the Processes section of the PCG part. Each terminal process communicates with its associated relay by a type Server communication. We say that terminal processes have one *communication port*, or simply *port*, of type Server. Relay processes may have any non-negative number of ports of Client and Prop types. We implement a solution with five terminal and three relay processes; relay processes have two ports of type Prop; relay[1] and relay[2] have two ports of type Client and relay[3] has one port of type Client. Having identified the processes and their ports, we specify in the Channels section of PCG the actual process topology by connecting ports, thus defining the channels of point-to-point communications. Ports are identified by unique integers within each type. For example, port 1 of type Server of terminal[1] (written as 'terminal[1].Server[1]') is connected (written as '<->'), with port 1 of type Client[1] of relay[1] (written as 'relay[1].Client[1]').

A program, called PCG-Builder, parses the PCG part of the script and produces the corresponding PCG. In PCGs, processes are depicted by two concentric circles and labeled by their name. On the inner circle, each type of dependency is indicated by a bullet from which lines to the outer circle are drawn. The meeting points with the outer circle denotes the ports of the type; ports in fig. 1 are indexed by their type initial (Server, Client, Prop) together with their unique integer within the type. A channel is represented by a line joining two ports.

The elements of the PCG described so far, specify a general application PCG independent of any particular MPE. In PVM3, messages are tagged by integer identifiers, tagids, which are used by the sending and receiving processes. The tagids annotate the arcs of the PCG. In the script, we specify the annotation of the arcs by giving them unique integer values (tagID: default). Nodes may optionally be annotated by host allocation information, if a process is to be spawned on a particular host. Finally, nodes are annotated by information for the sequential components: the full path name of the executables, as well as, any parameters required by the processes. In the script of fig. 1 we specify process allocation, give the names of the executables to be found in the default search path of PVM, we specify the integer parameters to terminal processes, being the values they hold. The maximum which all should receive is 999. The annotation of two nodes representing processes relay[1] and terminal[1] and the annotation of five channels is shown in fig. 1 (process names are replaced by integers, e.g. relay[1] is replaced by 1 and terminal[1] by 4). The PCG-annotator parses the second and third parts of the script and annotates the PCG.

```
Application AllToAllRelays;   PCG
Components
   terminal port-types: Server;
   relay    port-types: Client, Prop;
Processes
 relay[1], relay[2]    #ports= Client:2, Prop:2;
 relay[3]                    #ports= Client:1, Prop:2;
 terminal[1], terminal[2], terminal[3],
 terminal[4], terminal[5]      #ports= Server:1;
Channels
terminal[1].Server[1] <-> relay[1].Client[1];
terminal[2].Server[1] <-> relay[1].Client[2];
terminal[3].Server[1] <-> relay[2].Client[1];
terminal[4].Server[1] <-> relay[2].Client[2];
terminal[5].Server[1] <-> relay[3].Client[1];
relay[1].Prop[1] <-> relay[2].Prop[1];
relay[1].Prop[2] <-> relay[3].Prop[1];
relay[2].Prop[2] <-> relay[3].Prop[2];
```

PCG BUILDER



```
PARALLEL SYSTEM
Environment PVM3
PVM3_Annotation
   tagID : default;
PVM3_Options
 Allocation
 relay[1], terminal[1], terminal[2] at euridiki;
 relay[2], terminal[3], terminal[4] at kadmos;
 relay[3], terminal[5]              at lavdakos;

SEQUENTIAL COMPONENTS
Executable files
   terminal : path default file terminal;
   relay :    path default file relay;
Execution Parameters
 terminal[1]:6; terminal[2]:999;
 terminal[3]:7; terminal[4]:8; terminal[5]: 9;
```

PCG ANNOTATOR

```
Node 1  Name       : relay
       instance    : 1
       Allocation  : euridiki
       file        : relay
       path        : Default
       Parameters  : (None)
 Node 4  Name      : terminal
       instance    : 1
       Allocation  : euridiki
       file        : terminal
       path        : Default
       Parameters  : 6
Channel annotation
1 :4.Server[1]<->1.Client[1]
2 :5.Server[1]<->1.Client[2]
6 :1.Prop[1]<->2.Prop[1]
7 :1.Prop[2]<->3.Prop[1]
8 :2.Prop[2]<->3.Prop[2]
```

Fig. 1. Ensemble script, the produced PCG and part its annotation

2.2 The Design of Reusable PVM Program Components and the PVM Loader

Reusability of executable components in a message-passing environment demands that components should not assume any specific communication topology for the processes instantiated from them; instead they should specify an open communication interface. Establishing a point-to-point communication between two PVM processes two values are needed in each: the tid of the other process and the common tagid. We define a data structure, implementing a port, in which (tid, tagid) pairs are stored. A program component, in general, may have a number of ports of the same type, which are organized in an array. An executable component may have many types of ports. The types of ports form array Interface, the elements of which point to arrays of ports. Each port is now identified by an index to a type and a port index within the type. The parameters of communication routines (e.g. pvm_send) referring to tid and tagid are expressed as Interface[T].port[p].tid and Interface[T].port[p].tagid, where T and p are indices to a type and a port, respectively. The structure and the pseudo-code of the two components of the application are shown in fig. 2; they both declare Interface with terminal component having one and relay two communication types.

Upon their creation processes fix their interface; two actions are involved: a. creation of the appropriate number of ports for each type and b. setting value pairs (tid, tagid) to ports. The Loader visits the PCG nodes and spawns processes; all appropriate information annotates each node: the associate executable, its host allocation and its command line parameters; also the number of ports of each type are given as command line parameters. The first action of a process is to get the number of ports of each type and create the Interface structure. This is coded in the MakePorts routine. The value pairs (tid, tagid) for each port cannot, in general, be sent to the processes just spawned, as a process with which it needs to communicate may have not been spawned yet and its tid would not be known. The Loader annotates the PCG node with the tid of the process just spawned. Having visited all nodes and spawned all processes, the Loader visits the nodes of the PCG once more and sends the port information (tid and tagid) to the processes. The processes receive the port information and set their Interface. This activity is coded in the SetInterface routine.

```
void main(Int)              /* terminal */
  InterfaceType Interface[1];
                         /* 1= Server type */
{ MakePorts(Interface);
  SetInterface(Interface);
  realMain (Interface); }

void realMain (Interface);   { send
Int to port Interface[1].p[1]
receive GM from Interface[1].p[1]}
```

```
void main( )                          /* relay */
  InterfaceType Interface[2];
                 /* 1= Client and 2=Prop types */
{ MakePorts(Interface);
  SetInterface(Interface);
  realMain(Interface); }

void realMain (Interface);
{ receive values from all ports of Interface[1]
   find their local maximum LM
   send LM to all ports of Interface[2]
   receive LMs from all ports of Interface[2]
   find the global maximum GM
   send GM to all ports of Interface[1] }
```

Fig. 2. The common structure and the pseudo-code of terminal and relay

3 A Variation on the Distribution of Maximum Application

In the implementation of the previous section, all relay processes are connected with each other via their Prop ports, each exchanging with the others their local maxima. The same components will now be reused to compose a different design, in which relay processes form a tree topology; we use eight terminal and four relay processes. Relay[1] and relay[2] each have two Client ports connected with terminal[1], terminal[2] and terminal [3], terminal[4], respectively. The Prop ports of relay[1] and relay[2] are connected with Client ports 3 and 4 of relay[3], respectively. Relay[3] has four Client ports connected with terminal[5], terminal[6], relay[1] and relay[2]; the Prop port of relay[3] is connected to Client port 3 of relay[4]. Finally, relay[4], which is the root, has three Client ports connected with terminal[7], terminal[8] and relay[3] and no Prop port. The PCG part of the script and its graphical depiction are shown in fig.3. Let us note, that the Client and Prop types are compatible, as only one integer value is sent and received through them. At each level the relay processes receive the values from their client processes (terminal or relay), select their maximum and propagate it to their parent-relay. The root selects the global maximum and sends it to its client processes. The relay processes below the root do the same until the global maximum reaches all terminal processes.

```
Application TreeRelays;
PCG
Components
  terminal port-types: Server;
  relay    port-types: Client, Prop;
Processes
 relay[1], relay[2]#ports= Client:2, Prop:1;
 relay[3]          #ports= Client:4, Prop:1;
 relay[4]          #ports= Client:3, Prop:0;
 terminal[1], terminal[2], terminal[3],
 terminal[4], terminal[5], terminal[6],
 terminal[7], terminal[8]   #ports= Server:1;
Channels
 terminal[1].Server[1] <-> relay[1].Client[1];
 terminal[2].Server[1] <-> relay[1].Client[2];
 terminal[3].Server[1] <-> relay[2].Client[1];
 terminal[4].Server[1] <-> relay[2].Client[2];
 terminal[5].Server[1] <-> relay[3].Client[1];
 terminal[6].Server[1] <-> relay[3].Client[2];
 terminal[7].Server[1] <-> relay[4].Client[1];
 terminal[8].Server[1] <-> relay[4].Client[2];
 relay[1].Prop[1] <-> relay[3].Client[3];
 relay[2].Prop[1] <-> relay[3].Client[4];
 relay[3].Prop[1] <-> relay[4].Client[3];
```

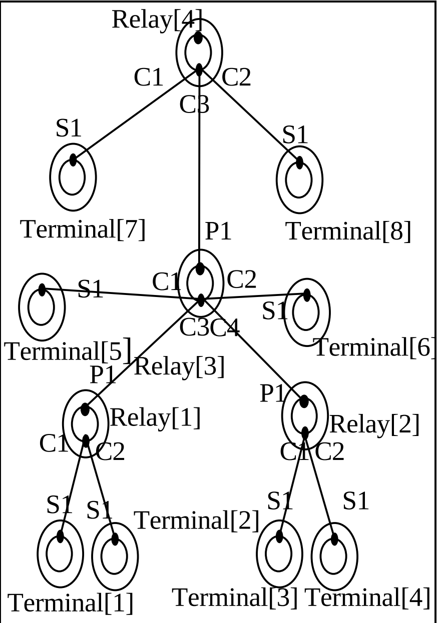


Fig. 3. The PCG part of the script and its graphic depiction

4 Conclusions

We have outlined Ensemble, an implementation methodology, applied to PVM programs with static process topologies. We demonstrated the flexibility of the methodology, by composing two solutions to the Distribution of Maximum application. In [3], more design variations are presented. We constructed reusable PVM library components, which may have any number of process instantiations either in the same or in other PVM programs; each instantiated process may have its own communication dependencies. The application topology is specified in scripts. By editing the scripts we scale programs, we change the allocation of processes to hosts or even change the topology of the processes, without modifying the program components.

Ensemble does not suggest a new MPE and does not demand any changes to MPEs, but acts like a shell to them. Ensemble does not cause any execution overhead, apart from the execution of the loader interpreting PCGs, which is negligible compared with the saved programmer’s development time for establishing topologies. Ensemble is independent of any design, visualization, performance, or any other tools which may be used with an MPE.

The methodology may be applied to any message-passing environment. We have applied it [2,4] on Parix [9] running on Parsytec GC3/512 and CC machines. Parix imposes altogether different constraints to programs than PVM, thus requiring a different PCG annotation, its own techniques of reusable program components and its own Loader. Ensemble implementations of the same design look similar in PVM and Parix. Also, the rewriting (porting at source level) of applications from one environment to the other is mechanical and in some cases may be automated. The Ensemble for MPI is currently under development. As MPI does not deal with dynamic process creation Ensemble does not constrain possible implementations, as in PVM.

Ensemble separates the elements of a message passing application: the process topology, the architecture, the mapping of the processes onto processors and the computations giving the required result or providing the service. The first three are described in the script and the last by the program components. As they are separated, they may be modified independently of one another, thus simplifying program debugging and maintenance. Script modifications permit rapid program variations, thus allowing to ask “what if” type of questions to test the performance and fine tune the application. Consequently, Ensemble “removes” from message passing implementations the aspects that most influence the structure of applications, namely process management. They are “removed” in the sense that it is not required to program process management, since process topologies are specified in the scripts. Consequently, Ensemble “removes” the effort of programming most of the architectural idiosyncrasies of message passing environments.

Ensemble provides a flexible and efficient means for composing applications. Although, the script language is still under development it was shown to be flexible and permitted the rapid composition of PVM programs. It is straight forward to edit scripts to either scale programs, in a regular or irregular ways, by adding and connecting new components, or to change the allocation of processes to processors, or to modify the topology, etc. This approach is related to the composition of object oriented applications by using objects and scripts [7], as it encourages a component oriented approach to application development. The Ensemble reusable components are used in the composition of applications in a “soft LOGO” manner. The programmer has only to code the sequential computations of applications.

PVM and MPI, support the portability of applications to a number of architectures meaning that programs may run without modifications on these architectures. Portability does not necessarily imply that applications run efficiently on any architecture. For a ported program to run efficiently it has to be adapted and fine-

tuned for performance. Implementations of programs with Ensemble do not only lose the portability offered by MPEs but also, provide the infrastructure for extensive structural changes, such as change of granularity of processes.
Future work includes high level parametric topology descriptions in scripts, support for dynamic process creation, tools for porting applications to different MPEs.

References

[1] F. Berman, L.Snyder, ‘On mapping parallel algorithms into parallel architectures’, J. P. Distr. Comput. 4, 5, 439-458.

[2] J.Y. Cotronis: A Methodology for Initiating Arbitrary Structured Programs in Parix by Interpreting Graphs, ZEUS 95, Parallel Programming and Applications, ed. P. Fritzon and L. Finmo, IOS Press 1995.

[3] J.Y.Cotronis: Efficient composition and automatic initialization of arbitrary structured PVM programs, in IFIP Proceedings of the 1st Workshop on Software Engineering for Parallel and Distributed Systems, (held in association with ICSE 96), Berlin, March 1996.

[4] J.Y.Cotronis: Efficient Program Composition on Parix by the Ensemble Methodology, 22nd Euromicro Conference 96, Prague.

[5] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, Vaidy Sunderam, ‘PVM 3 User’s guide and Reference Manual’, ORNL/TM-12187, May 1994.

[6] MPI: A Message-Passing Interface Standard, Message Passing Interface Forum, June 12, 1995.

[7] O. Nierstratz, D. Tsichritzis, V. de Mey, M. Stadelmann, ‘Objects + Scripts = Applications’, in Proceedings, Esprit 1991 Conference, Kluwer Academic Publishers, 1991, pp. 534-552.

[8] M.G.Norman, P.Thanisch, ‘Mapping in Multicomputers’, ACM Computing Surveys, Vol25, No.3, Sept. 93.

[9] Parix, Manuals Parix 1.2, 1.9, Parsytec Gmbh.

[10] P.Pouzet, J.Paris, V.Jorrand, ‘Parallel Application Design: The Simulation Approach with HASTE’, Proc. High Performance Computing and Networking, Munich, April 18-20, 1994, Vol II, pp. 379-393.

[11] C. Scheidler, L.Schaefers, ‘TRAPPER: A Graphical Programming Environment for Industrial High-Performance Applications’, PARLE Conf., Munich, 403-413, 1993.