In Proc. 6th Euromicro Workshop on Parallel and Distributed Processing, IEEE Computer Science 1998, pp. 409-415

Porting Applications between PVM and Parix under Ensemble

J.Y. Cotronis, E. Avgeri and Y. Krallis
Dept. of Informatics, Univ. of Athens, Panepistimiopolis, 157 71 Athens, Greece
Tel.: +30 1 7291885 fax. +30 1 7219561 e-mail: cotronis@di.uoa.gr

Abstract

We examine the differences between two message passing environments, PVM and Parix, and their impact on the implementation of applications. We outline Ensemble, a methodology for designing and implementing message passing applications by providing common software architecture. We outline Ensemble applied to PVM and Parix. We describe the mechanical porting of applications developed under Ensemble from one environment to the other.

1. Introduction

The most widely used programming paradigm for distributed memory systems is message passing. Message passing interfaces [6] introduced by vendors relied on and exploited the special characteristics of their architectures. The implementation of message passing designs was difficult, as they depended on knowledge of the actual hardware platform. In the past few years, we have seen the emergence of Message Passing Environments (MPE), such as PVM [5], Parix [8], MPI [7], in which message-passing applications may be implemented without the precise knowledge of the underlying architecture. MPEs, particularly PVM and MPI, abstract architectural characteristics and permit application implementation on various platforms. Applications are thus portable among the parallel systems for which the interface is available.

However, the implementation of application designs on MPEs is still a demanding task. The application implementation does not only involve the programming of process computations, but also requires the explicit programming of process management issues, such as their creation, their logical topology and their mapping onto the architecture. Consequently, the original design is diffused into the implementation, as statements implementing computations and process management co-exist in code segments. Furthermore, code development has to anticipate for all possible occurrences of the processes; for example all their possible positions in application topology and their interactions with their neighbouring processes in each case. The implementation complexity increases when the topology is not regular, as process positions and their communicating processes cannot be determined by functions.

There are no methodologies supporting the implementation of application designs. In addition, reusability of executables is restricted as process management is encoded in them. Processes may only operate within the specific topology for which they are implemented, although their functionality could be used in other applications. MPEs may differ significantly in the way they manage processes, but also in other fundamental issues, such as the kind of processes they support (threads, lightweight and heavyweight, etc.), the naming and identification of processes. Due to these differences, some applications are easier to implement on some MPEs and more difficult on others. For example, a general tree topology is relative easy to implement on PVM, but more demanding on MPI or Parix. Each MPE requires its own implementation techniques and imposes its own structure on message passing implementations. Two programs implementing the same design, using the same sequential language, but implemented on different MPEs, look very different. Porting applications from one MPE to another is, in general, costly, as it is tackled case by case.

We have developed a message passing implementation methodology, called Ensemble, in which implementations on any MPE have common software architecture. They preserve the original design and, for this reason, are easier to develop, debug and maintain; they are also easily portable from one MPE to another. Ensemble is not “yet another” message passing environment. It is a methodology, independent of any MPE, aiming to reduce the cost of message passing software development, maintenance and porting to other MPEs. In [2,4] we presented Ensemble for PVM and in [3] for Parix. In this paper, we present Ensemble comparatively for PVM and Parix and demonstrate the mechanical portability of applications between PVM and Parix. The structure of the paper is as follows: In section 2, we outline the characteristics of PVM and Parix and we examine their implications on application implementations. In section 3, we present the common software architecture of Ensemble implementations on PVM and Parix. In section 4, we address the portability of applications between PVM and Parix. Finally, in section 5 we present the conclusions.

2. PVM and Parix: two very different MPEs

We outline the main characteristics of PVM and Parix.

2.1 PVM overview

1. The underlying architecture of PVM is any host system running UNIX and some special cases of parallel architectures, which are viewed as virtual machines.
2. The PVM console allows the user to interactively start, query, and modify the virtual machine. Any PVM program may use the complete host system.
3. A PVM process is a UNIX process running on a host machine. A process is spawned by its parent process. To run PVM programs the user spawns a root process.
4. Processes are identified by unique task identifiers (tids) generated upon their creation. The tid is only known to the parent process; the spawned process may obtain its parent's and its own tid by function calls.
5. Processes may be spawned on specific hosts. If no host is specified, PVM chooses where to spawn them.
6. Process communication and synchronisation are of two categories: a) Requiring process tids and possibly some message tag identifiers (tags), such as point-to-point asynchronous communication (pvm_send, pvm_recv, etc.) and multicast, sending the same value to a list of processes. In that sense (tid, tagid) pairs are the primary communication parameters. b) Requiring group definitions, such as bcast, sending the same value to processes in a group, and barriers, for synchronising processes in a group. Programming applications forming, in general, tree-like process communication dependencies, where each process communicates only with its parent and its children, is easy to program. However, establishing general graph process topologies requires substantial programming effort.

2.2 Parix overview

1. Parix runs on PARSYTEC architectures and views them as a logical grid of processors. The fist version was developed for transputer grids. Later versions however, run on super cluster architectures based on nodes running AIX.
2. Throughout program execution a partition of processors is exclusively reserved and managed as a private resource of the application program.
3. Applications are initiated from a front-end computer by loading the same initial main program onto all processors of the application partition. Thus, a Parix program initially appears as an SPMD program, each copy having its own context, i.e. an environment with its own code and data. A set of global data kept at each processor allows identification of the "own" processor position within the network. Depending on the position it is possible to execute different sections of the main code or execute identical instructions on different data. Contexts cannot migrate to another processor.
4. A program may create lightweight processes (threads) handling asynchronous services. Threads are running concurrently in the same context and share all global variables defined in the program. Semaphores are provided for variable protection and synchronisation.
5. It is also possible to load and run a different code on a processor by issuing an Execute call. This call loads an executable Parix program (on the processor the calling thread runs on) and creates a new context, distinct from the context of the calling thread. The calling thread waits for termination of the new context before it resumes execution. More than one context may run on the same processor. Thus, an application, which initially looks like an SPMD, may become during run-time a true MIMD.
6. Communication between threads may be synchronous (S) or asynchronous (A). Communication may be based on virtual links (L), which build point-to-point connections between threads. A set of virtual links can be combined to build a virtual topology (T). Communication may also be random (R), that is, not requiring the definition of virtual links or topologies. There are routines for sending and receiving messages implementing the above communication types, as shown in the following table:

	Links (L)	**Topology (T)**	**Random (R)**
Synchronous (S)	SendLink RecvLink	Send, Recv	SendNode RecvNode
Asynchronous (A)		ASend, ARecv	(PutMessage GetMessage)

Note that asynchronous communication over links is not supported. In Parix 1.9 AR communication type is also not supported and for this reason are placed on parentheses.

7. Communication types define, explicitly or implicitly, communication channels between threads. When virtual links are used, the channels are explicitly defined. A link between two threads is established when each calls a connection routine (e.g. ConnectLink) giving as parameters each other's Processor Identifier, ProcId, and a common Request Identifier, ReqId. A topology groups links under a name and gives them unique symbolic names within the topology, thus defining channels explicitly and abstractly. In random communication, the channels are implicitly specified by referring directly to the processor identifiers the two threads are running on and to the common ReqId. Consequently, in all communication types between two threads the primary communication information is the ProcIds of the threads and the ReqIds tagging the messages over the channel. As threads cannot migrate, the processor identifier specifies the processor allocation of the thread and the request identifier uniquely specifies a particular channel.

2.3 Implementation on PVM and Parix

We will demonstrate the implementation on PVM and Parix, using an application, called Distribution of

Maximum. There are two types of processes, *terminal* and *relay*. Each terminal process accepts an integer parameter and requires the maximum of these integer parameters. Groups of terminals send their values to a relay process (acting as their server). Relays find the local maximum of their associated terminals, which they asynchronously send to the other relays and, respectively, they receive the local maxima from them. Each relay finds the maximum value, which they send to their respective terminal processes.

Terminal processes have one communication dependency, that with their associated relay processes, which we call S (server) type. Relay processes have two types of communication dependencies, one with their terminal processes, which we call C (client) type, and one with the relay processes, which we call P (Pier-to-pier) type. A relay process may have any non-negative number of dependencies of C and P types. Processes are depicted by two concentric circles, as in fig. 1. On the inner circle, the communication types are depicted, namely S for terminal processes and C, P for relays. On the outer circle, the specific number of interconnections or ports of each communication type are depicted. Type S has one port (fig. 1a) and C, P have n, m ports, respectively (fig. 1b).

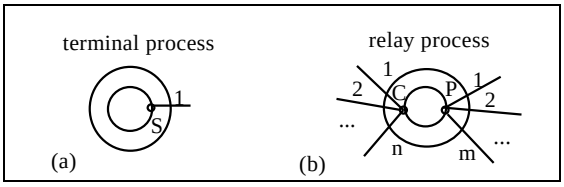

Figure 1: Representation of processes

The application topology with three relays, each one connected with two terminals, is depicted in fig. 2. Graphs are used as a natural representation of process communication [1]. Arcs connecting ports represent the communication channels between processes.

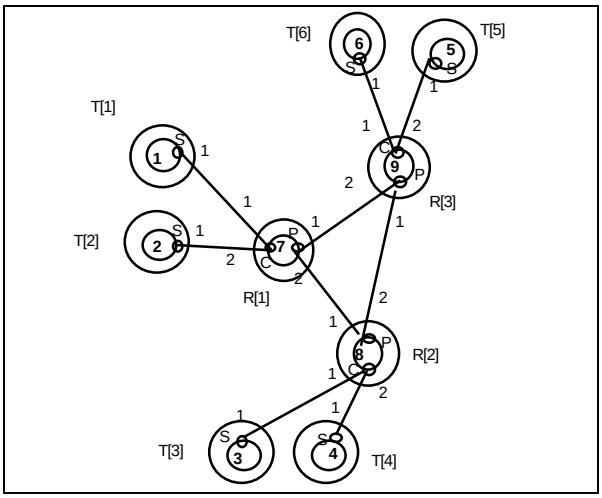

Figure 2: Topology of Distribution of Maximum

The PVM implementation style we present is not unique, but typical for PVM and may be easily generalised for any application. A root-process spawns all terminal and relay processes and stores their tids. Then it sends to each process the tids of the processes it needs for communication. Processes are responsible for interpreting the tids (integer values) they receive in a compatible manner. In the outline of the root-process that follows, we use a simplified PVM spawn instruction, which defines the executable from which a process (e.g. terminal) is started, the parameters of the executable (e.g. “15”) and the host which will run the process (e.g. gaia). We also use a simplified Send instruction, which has the following structure: Send (Where: V1,V2,…,Vn). The parameter Where is the tid of the process that the message is sent to and V1, V2,…, Vn the data of the message sent. These syntactic modifications are used for brevity, as well as for identifying similarities and differences of PVM and Parix.

```
                    /* PVM Root Process */
/* Create processes and store tids */
                              /* Spawn terminals */
TerminalTid[0]=spawn(terminal,”15”, gaia);   /* Node 1 */
TerminalTid[1]= spawn(terminal,”34”, gaia);  /* Node 2 */
TerminalTid[2]= spawn(terminal,”37”,chaos); /* Node 3 */
TerminalTid[3]= spawn(terminal,”4”,  chaos); /* Node 4 */
TerminalTid[4]= spawn(terminal,”5”,  eros);   /* Node 5 */
TerminalTid[5]= spawn(terminal,”99”, eros);  /* Node 6 */

                              */ spawn relays */
RelayTid[0]= spawn(relay,” “, gaia);          /* Node 7 */
RelayTid[1]= spawn(relay,” “, chaos);         /* Node 8 */
RelayTid[2]= spawn(relay,” “,eros);           /* Node 9 */

/* Send tids to establish connections */
Send(TerminalTid[0]: RelayTid[0]);
Send(TerminalTid[1]: RelayTid[0]);
Send(TerminalTid[2]: RelayTid[1]);
Send(TerminalTid[3]: RelayTid[1]);
Send(TerminalTid[4]: RelayTid[2]);
Send(TerminalTid[5]: RelayTid[2]);

Send(RelayTid[0]:RelayTid[1],RelayTid[2],
                    TerminalTid[0],TerminalTid[1]);
Send(RelayTid[1]:RelayTid[0],RelayTid[2],
                    TerminalTid[2],TerminalTid[3]);
Send(RelayTid[2]:RelayTid[0],RelayTid[1],
                    TerminalTid[4],TerminalTid[5]);
```

In Parix, the same main program runs on a partition of processors reserved by the application. If each process runs on a unique processor, we need nine (9) processors uniquely identified by ProcIds from 0 to 8. The outline of the main program, which according to Parix is initiated on each of the nine processors, is shown below. Each process gets the ProcId it runs on and loads the appropriate executable and command line parameters.

```
                    /* Parix Main Program */
MyProcID = Get_my_ProcId;
case MyProcID of
{ 0 : Execute(terminal, 6,"15");          /* Node 1 */
  1 : Execute(terminal, 6,"34");          /* Node 2 */
  2 : Execute(terminal, 7,"37");          /* Node 3 */
  3 : Execute(terminal, 7,"4");           /* Node 4 */
  4 : Execute(terminal, 8,"5");           /* Node 5 */
  5 : Execute(terminal, 8,"99");          /* Node 6 */
  6 : Execute(relay, 7,8,0,1);            /* Node 7 */
  7 : Execute(relay, 6,8,2,3);            /* Node 8 */
  8 : Execute(relay, 6,7,4,5);            /* Node 9 */
}
```

We used the Execute(executable, P1, P2, …, Pn) which spawns a process from "executable", passing parameters, P1, P2,…, Pn. We have put application parameters in quotes to simplify the comparison with PVM implementation. The other parameters are the ProcIds of the processes with which each process needs to communicate. For example, the first and second terminal processes (ProcIds 0 and 1) have parameter 6 which means that they will communicate with (relay) process with ProcId=6. Relay with ProcId=6 has parameters 7, 8, 0, 1, which means that it communicates with the relays having ProcIds 7 and 8 and with the terminals having ProcIds 0 and 1. Executables are responsible for interpreting their parameter list in a compatible manner.

Executables terminal and relay, used in each system, have again differences. We will show them, as well as their similarities. We have extended the notation Send(Where: P1, P2, …, Pn) for receive instructions, Receive(Where: P1, P2, …, Pn). Furthermore, we have used as Where variable names of communication ports S1, C1, C2, P1, P2 used in the process communication graph of fig. 2. The type and values of port variables are different for PVM and Parix. In the coding of terminal and relay that follows, we localise and hide these differences in Initial Actions:

/* Terminal */

```
void main (argc, argv);
{ Initial Actions;
    GetParam(V); Send(S1: V); Receive(S1: GMax);}
```

/* Relay */

```
void main (argc, argv);
{ Initial Actions;
    LMax=0;
    Receive(C1: V); if (V>LMax) LMax=V;
    Receive(C2: V); if (V>LMax) LMax=V;
    Send(P1: LMax); Send(P2: LMax);
    GMax= LMax;
    Receive(P1: V); if (V> GMax) GMax=V;
    Receive(P2: V); if (V> GMax) GMax=V;
    Send(C1: GMax); Send(C2: GMax); }
```

In PVM, terminal processes receive the tid of the relays that communicate with, and relay processes receive the tids of two relays and two terminals from the root-process. The Initial Actions of terminal and process in PVM are:

Terminal Initial Actions	Relay Initial Actions
Parent = get_parent_tid;	Parent = get_parent_tid;
Receive(Parent:S1);	Receive(Parent: P1, P2, C1, C2);

According to the Parix implementation policy we have adopted, each process gets the ProcIds of the processes it communicates from its parameters by calling a GetParam routine. However, Parix supports different kinds of communications (Random, Links and Topology). If we choose Random communication, we skip any further action. We describe further actions for communication over Links. For two processes to establish a link, each one must make a ConnectLink call, using as parameter the ProcId of the other process. ConnectLink returns a link, which is used by send and receive routines thereafter. The execution of ConnectLink enforces synchronous communication between the processes; consequently, the danger of deadlock is apparent. If for example, we alter the order of parameters 6, 8 in ProcId=7, i.e. from Execute(relay, 6,8,2,3) to Execute(relay, 8,6,2,3), the program would deadlock. Relay with ProcId=6 would wait to be connected with relay with ProcId=7, which would wait to be connected with relay with ProcId=8, which in turn would wait with relay with ProcId=6, forming a dependency cycle. As the order of the parameters stands now there is no dependency cycle.

Terminal Initial Actions	Relay Initial Actions
GetParam(R);	GetParam(R1, R2, T1, T2);
S1=ConnectLink(R);	C1=ConnectLink(T1);
	C2=ConnectLink(T2);
	P1=ConnectLink(R1);
	P2=ConnectLink(R2);

Although we have simplified and hidden a number of differences of PVM and Parix, the implementations of the application are dramatically different.

3. The Ensemble methodology

Structuring of implementations described in the previous section, as an organiser and executable programs, leads to two observations, regarding reuse of executables and generalisation of establishing process topologies.

Reuse of executables. The Distribution of Maximum implementation on the two MPEs may be easily scaled by adding to the root-process in PVM and to the main in Parix, instructions for the creation and connection of terminal and relay processes. Changes to the executables of terminal and relay are not required. They are like library routines. We observe that they don't involve any instructions for topology or any process creation. Before any calculations are performed by these processes, the

communication is established with the use of the instructions included in their respective Initial Actions.

However, there is a limitation; all relays must have exactly two terminals as clients. Usually, due to programming complexity, scalability is designed on global factors in an application, e.g. sizes of dimensions of a grid topology, or relays having a specific number of terminals. However, there may be other local scalability factors. In general, scaling of applications requires replication of processes and their interconnections. For some process topologies, such as a torus, it is sufficient to replicate identical processes each having the same number of connections. However, for other topologies, such as master/slave, each replicated process may have a distinct number of interconnections, possibly within a range. We would like to permit the design option to scale the application by adding any number of terminals to any relay, so that relays may have any number of terminals.

Generalisation of establishing process topologies. We observed that the topology, which is created by the root-process of PVM and the main program of Parix, is described by the general graph of the application of fig. 2. We may annotate this graph by information pertinent to PVM or Parix. For example, all parameters of spawn may annotate the corresponding nodes. For Parix implementation, nodes may be annotated with the associate ProcIds. The arcs of the graph may be annotated by tagids and ReqIds of PVM and Parix messages, respectively, which have been ignored for simplicity.

Since all information needed to implement a topology, appears on the application graph, the root-process for PVM and the main routine for Parix may be generalised to launcher programs or loaders of the application.

In Ensemble, application implementation is an 'ensemble' of an annotated graph, the reusable and scalable executables and a launcher program. Each MPE requires its own graph annotation, its own techniques for reusable executables and its own launcher. We have developed these tools for PVM [2,4] and Parix [3]. We briefly describe the implementation of the Distribution of Maximum using Ensemble.

3.1 The annotated graph.

We have developed a script language, which describes annotated graphs of applications. A script abstractly describes the processes and their communication channels, the mapping of the processes onto the architecture, the executables and the application parameters. The script for the Distribution of Maximum application is shown in fig. 3. The script is structured in two main parts:

Ensemble Script

Application Distribution_of_Maximum;
PCG
Components
terminal port-types: S[1..1];
relay port-types : C[1..], P[0..];
Processes
relay[1],relay[2], relay[3] #ports = C:2, P:2;
terminal[1], terminal[2], terminal[3],terminal[4], terminal[5], terminal[6] #ports=S:1;
Channels
terminal[1].S[1] <-> relay[1].C[1]; terminal[2].S[1] <-> relay[1].C[2];
terminal[3].S[1] <-> relay[2].C[1]; terminal[4].S[1] <-> relay[2].C[2];
terminal[5].S[1] <-> relay[3].C[2]; terminal[6].S[1] <-> relay[3].C[2];
relay[1].P[1] <-> relay[2].P[1]; relay[1].P[2] <-> relay[3].P[1]; relay[2].P[2] <-> relay[3].P[2];

PARALLEL SYSTEM
PVM3
tagID : default;
Process Allocation
relay[1], terminal[1], terminal[2] on gaia;
relay[2], terminal[3], terminal[4] on chaos;
relay[3], terminal[5], terminal[6] on eros;
Executable Components
terminal: path default file terminal.sun;
relay : path default file relay.sun;
Parameters
terminal[1]:"15"; terminal[2]:"34"; terminal[3]:"37";
terminal[4]:"4"; terminal[5]:"5"; terminal[6]:"99"

PARALLEL SYSTEM
PARIX [5]
reqID : default;
ProcID : default;
Communication Type R

Executable Components
terminal: path default file terminal.px;
relay : path default file relay.px;
Parameters
terminal[1]:"15"; terminal[2]:"34"; terminal[3]:"37";
terminal[4]:"4"; terminal[5]:"5"; terminal[6]:"99"

Figure 3. The Script for application Distribution of Maximum for PVM and Parix

PVM Reusable Components	Parix Reusable Components
#include Libraries for PVM and Ensemble PVM port={int tid, tagid}	#include Libraries for Parix and Ensemble Parix port={int ProcId, Reqid; link L; Top T}

```
/* common main */        void main(argc, argv);
                         {  struct port_type {int portcount; port_struct *port}
                            type_def struct *port_type Interface;
                            extern int TypeCount;
                            Makeports(Interface); SetInterface(Interface); RealMain(Interface)}

/* Terminal RealMain */  void RealMain (Interface, argc, argv);
                         { Int TypeCount=1; GetParam(V); Send(S, 1: V); Receive(S, 1: Max); }

/*  Relay RealMain */    void RealMain (Interface, argc, argv);
                         { Int TypeCount=2;
                           LMax=0;
                           for (i=1; i=C.portcount; i++)  { UReceive(C, i: V);  if (V>LMax) LMax=V};
                           for (j=1; j=P.portcount; j++) { USend(P, j: LMax); }
                           GMax= LMax;
                           for (j=1; j=P.portcount; j++) { UReceive(P, j: V); if (V>GMax) GMax= V};
                           for (i=1; i=C.portcount; i++) {USend(C, i: GMax)} }
```

Figure 4. The common structure of reusable and scalable terminal and relay

The first part, headed by PCG, specifies the Process Communication Graph (PCG) of applications independent of any MPE. PCGs are a natural structure for specifying processes and their communication dependencies and are close to program design. Nodes on a PCG denote processes and arcs communication channels between them. In the PCG part, we first specify the components involved (e.g. T and R) together with the number of ports of each communication type. Then we specify the processes instantiated from each component, together with the actual number of ports of each communication type. Finally, we specify the communication channels between processes.

The second part, headed by Parallel System, includes information for the annotation of the PCG according to the implementation environment. For a PVM script, the tagid annotation must be specified and optionally the allocation of processes. For a Parix script, the number of processors allocated to the application should be specified, as well as the mapping of processes to processors. In Parix, the communication type (R, L and T) should also be specified. Finally, information about the executables (name and full path) and the parameters of each process are specified.

The PCG-builder program common for all MPEs, reads the PCG part of the script and builds the PCG. The PCG-annotator specific for each MPE, reads the second part of the script and annotates the PCG graph.

3.2 The reusable program components

These are programs for calculating results or providing the service of the application. They do not include any process management. A message passing application is composed of processes, spawned from executables, which have open interfaces, do not assume any particular topology or any particular processes for communication. Their open interface consists of arrays of ports, one array for each communication type. Each port stores information needed for the communication routines. For PVM, a port is a (tid, tagid) pair. In Parix, a port is also a pair (ProcId, ReqId) and possibly a Link variable L and a Topology name T. Although port information differs in MPEs, the structure of Interface is the same (fig. 4). The first action of a process is to obtain the number of ports of each type from its parameter list and fix Interface reserving space for the ports for each type. This is coded in the MakePorts routine. Port information, (tid, tagid) for PVM and (ProcId, ReqId) for Parix, is then obtained and stored in Interface. This activity is coded in the SetInterface routine. An interesting case is the creation of Links that demands synchronisation of ConnectLink calls. For the executable program components to be reusable, the danger of a deadlock creation should be avoided. For this reason, we used one thread for each connection, which provides an asynchronous connection of processes. Finally, RealMain routine is called, where the application activities are coded. The programmer has only to use this common structure and program the RealMain routines of the applications. In the Distribution of Maximum application, we need coding of terminal and relay. Where parameter of Send and Receive refers to ports (CommunicationType, PortIndex). We have also written universal send and receive routines, USend and UReceive, respectively, which call appropriate PVM and Parix routines, making RealMain identical in PVM and Parix.

3.3 The launcher

The launcher is a program, which interprets the annotated PCG and composes applications. It is different

for each MPE, but universal for all applications implemented under the same MPE.

PVM launcher acts in two phases. In phase 1, it visits nodes of annotated PCGs, spawns processes with command line parameters the number of ports. It stores the tids of spawned processes. In phase 2, it visits nodes and sends port information to set their interface. PVM launcher is an extension of the root-process of section 2.

Parix launcher runs on all processors, visits PCG nodes, spawns processes and passes the appropriate parameters needed for Execute calls. Parix launcher acts in one phase, as all port information is known at compile time. Parix launcher is an extension of Parix main of section 2.

4. Portability between PVM and PARIX

Portability of applications between PVM and PARIX requires porting of script and of program components. The two implementations have already the same structure. Furthermore, script and reusable components have a number of common parts.

4.1 Porting of script

The PCG part of the script, that specifies the Process Communication Graph of the application, is exactly the same for both PVM and PARIX implementations, as it specifies the topology of the application and is independent of the implementation's environment. For the Distribution of Maximum script, refer to fig. 3.

The second part of the script, headed by PARALLEL SYSTEM, includes elements that specify the annotation of the PCG and depends on the implementation environment. The default function that specifies the tagid and reqid is the same in both environments and annotates the arcs of the PCG with unique positive numbers. In PVM there is an optional part, headed by Process Allocation, where processes are allocated to hosts, but in PARIX, we must specify ProcIds of processes. In PARIX, the default function, that specifies the mapping of processes to processors, uses the modulo of the number of processors. In order to port a PVM application to PARIX, we correspond each PVM host to a PARIX processor and define their number as the number X of the processors (Parix[X]). For porting from PARIX to PVM there are two alternatives. We can drop the Process Allocation part, as it is optional, or we can assign processes to PVM hosts.

In the part of the script that refers to the executables, we specify their name and the location, as well as the parameters of the processes. If these parameters are constants, there is no change at all.

4.2 Reusable components

Ensemble handles reusable components in the same way in both systems. Each component uses a library that includes MakePorts, SetInterface, USend and UReceive routines, that are environment depended. Main is the same in PVM and Parix; it calls MakePorts, SetInterface and finally RealMain.

RealMain, where all process communications and computations are performed, is also the same. The communication routines are called in an abstract way from USend and UReceive referring to communication ports. By this method, the portability of source code from one system to the other is direct, requiring compiling of RealMain and their linking to main.

4.3 Portability and communication types

PVM supports asynchronous communication, point to point and broadcast, whilst PARIX has five communication types: SL, ST, SR, AT, AR (section 2), all point to point. Direct porting from one system to the other is possible using the communication type Parix-AR and PVM-point-to-point. Using Ensemble, porting from an asynchronous type of Parix to another is very simple. We just need to define in the script, the required communication type, e.g. R, L or T. Porting from AR to AT and the reverse is automatic. Converting synchronous programs to asynchronous may be also mechanical, but needs care as communication dependencies change. The programmer must ensure that the application doesn't lead to a deadlock.

5. Conclusions

We examined the differences between PVM and Parix and their implications on the structure of applications. We outlined Ensemble and its common software architecture of implementations in PVM and Parix. We described the mechanical porting of implementations under Ensemble from one system to the other. The mechanical porting between MPEs is very important, as it guarantees that applications developed on an MPE may be ported to future MPEs with the minimum and predictable effort.

References

[1] G.R.Andrews: Paradigms for Process Interaction in Distributed Programs, ACM Computing Surveys, Vol.23, No1, March 91.

[2] J.Y.Cotronis: Efficient composition and automatic initialization of arbitrary structured PVM programs, Proc. 1st Workshop on PDSE, ICSE 96, Berlin, March 96.

[3] J.Y.Cotronis: Efficient Program Composition on Parix by the Ensemble Methodology, Euromicro96, Prague, 1996.

[4] J.Y.Cotronis: Message-Passing Program Development by Ensemble, Proc. PVMPI97, M.Bubak, J. Dongarra, J. Wasniewski (Eds.) LNCS 1332, pp 242-249, November 97.

[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] O.A. McBryan, "An overview of Message Passing Environments", Parallel Computing 20 (1994) 417-444.

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

[8] Parix1.2, Manual.