Efficient Program Composition on Parix by the Ensemble Methodology

J.Y. Cotronis
Dept. of Informatics, Univ. of Athens, Panepistimiopolis, TYPA Buildings, 157 71 Athens, Greece.
e-mail: cotronis@di.uoa.gr Phone: +30 1 7291 885 Fax: +30 1 7219 561

Proceedings of 22nd EuroMicro Conference, edit. P. Milligan and Kuchcinski, IEEE Computer Society Press, pp 545-552

Abstract

A message passing program composition methodology, called Ensemble, applied for Parix is presented. Ensemble overcomes the implementation problems and complexities in developing applications in message passing environments. Parallel applications are virtually specified by Process Communication Graphs (PCGs) annotated with communication information for Parix processes. Annotated PCGs are generated from application scripts by supporting tools. Reusable Parix executable components are defined from which all processes are created. A universal Parix program loader interprets the annotated PCGs creating the application processes from the reusable components and establishing their communication dependencies. Ensemble is applied to compose variations of Parix applications using the same reusable components. The methodology has been applied for PVM.

Keywords: software engineering for parallel systems, message passing program composition, reusable message passing components, annotated process communication graphs, message passing scripts, Parix.

1. Introduction

Message passing programming environments, such as Parix [8] and PVM [5], influence the software architecture of applications to such a degree that two programs implementing the same design (e.g. a simple ring topology) developed in two different environments, both programmed in, say, C with library message passing extensions, appear to be dramatically distinct. Typical environment characteristics influencing the design of applications are: 1) How is the application initiated, 2) The structure of processes, 3) How and where processes are created, 4) The process identification. Curiously, the message passing routines themselves, that is, how processes communicate, interact, synchronize, etc. in each environment do not influence the architecture of the applications but they have a rather more local influence.

Message passing environments allow for a general form of MIMD parallel computation, as application programs may possess arbitrary control and dependency structures. At any point in the execution of a concurrent application, the processes in existence may have arbitrary relationships between each other and in addition, any process may communicate and/or synchronize with any other. As in all programming environments, however, there are application types that are well suited to the characteristics of particular environments making them easy to implement and application types that are not so well suited and are thus much more difficult to implement.

In this paper we present the Ensemble methodology for implementing message passing program designs on Parix by the efficient composition of reusable message passing components. Ensemble has been applied for PVM [5] and although, the methodology is in principle the same for all message passing environments, the development of supporting techniques and tools required for each, differs significantly and demands specific treatment. It is our intention to develop a methodology in which implementations of the same program design in different message passing environments look similar and are implemented efficiently.

Let us examine the Parix characteristics which influence the application architecture.

1.1 Parix overview

Parix runs on PARSYTEC architectures and views them as a logical grid of processors. Throughout a program execution a set of processors, a partition or a network of processors, is exclusively reserved and managed as a private resource of the application program.

Applications are initiated from a front-end computer by loading the same initial main application program on all processors of the partition. The main application program is an executable residing in the file space of the front-end machine. Thus, a Parix MIMD program appears as an SPMD program. A set of global data kept at each processor allows identification of the “own” processor position within the network. Depending on this position it is possible to execute different sections of the main code or execute identical instructions on different data. A complete application may well be the cooperating copies of this main program.

A program may create light-weight processes, called also threads, handling asynchronous services. Threads are running concurrently in the same context, i.e. an environment with its own code and data, and share all global variables defined in the program. Variable protection and synchronization between threads in the same context may be achieved by using semaphore operations. Contexts cannot migrate to another processor. Thus an application which initially looks like an SPMD application may, during run-time, become a true MIMD, as local threads are created in contexts.

It is also possible to load and run an altogether different code at any time 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 completely 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 contexts may run on the same processor each being issued by a different thread.

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:

	synchronous (S)	*asynchronous (A)*
virtual links (L)	SendLink, RecvLink	
topology (T)	Send, Recv	ASend, ARecv
random (R)	SendNode, RecvNode	(PutMessage, GetMessage)

There are five communication possibilities, denoted hereafter by SL, ST, SR, AT, AR from the corresponding column (S or A) and row (L, T, R). Note that there is no AL communication routines, that is asynchronous communication over links is not supported. In Parix 1.9 running on CC machines AR are not supported either; for this reason routines are placed in parentheses.

Communication types define, explicitly or implicitly, communication channels between threads. All programming notations based on message passing between processes provide channels of some form and primitives for sending to and receiving from them [1]. A channel is an abstraction of the physical communication network in that it provides a communication path between threads. When virtual links are used the channels are explicitly defined by point-to-point links between threads. 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. Topologies group links under a topology name and give them unique symbolic names within a 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 the unique ReqId.

Consequently, in all communication types between two threads the *primary communication information* are the ProcIds of the participating threads and the ReqIds tagging the messages over the channel. As threads cannot migrate the processor identifier specifies the processor position of the thread and the request identifier uniquely specifies a particular channel of communication, as there may be more than one thread running on a processor, or two threads may communicate by more than one channel. Random communication directly uses the primary communication information; communication over links needs it to define the links; and topology communication uses links to define topologies.

1.2 Applications in Parix

Applications of the SPMD paradigm are easy to implement in PARIX. All processors are loaded with the same main program and each, depending on its position, operates on a different set of data. Also, well-structured applications which consist of the same program component loaded in all processors, each communicating with others is some regular way (e.g. ring, grid, complete binary tree, hypercube, etc.) is well suited to the Parix model. For these cases library routines have been defined establishing virtual topologies relieving the programmer from having to built them. By regular we mean that there are functions usually depending on ProcIds which determine the communication dependencies.

However, programming arbitrarily structured application programs is not, in general, an easy task in PARIX. For example, when the communication dependencies of program components does not form complete binary trees. In this case the same program component is loaded on all processors, as before, but as the communication dependencies are not regular (there cannot be general functions determining communication of arbitrary tree nodes) the programmer has to program the particular communication dependencies individually. It is even more difficult when different types of program components are required to be loaded and the communication dependencies between them form arbitrary graphs. Establishing arbitrary structured graph-like process communication dependencies *requires a substantial programming effort*. The effort is twofold:

(i) Programming the loading of the various program components. The main program has to load the appropriate program components, depending on the processor's position. The programmer has to write the "main" application program as well as the programs to be loaded later. Implementations *burden the design of the distributed application*.

(ii) Programming of the communication dependencies between program components directly in the source code. The *reusability of the program components is limited* since such components rely on the specific dependencies (ProcId, ReqId) they are involved in and they cannot be used without modification to establish a different communication dependency structure.

1.3 Ensemble methodology overview

The Ensemble methodology comprises three facets:

1. The Process Communication Graphs (PCG), which are used as a natural structure for specifying the processes and their communication dependencies. PCGs are close to the program design. Nodes on a PCG denote processes and arcs the communication dependencies between them. PCGs have been used in modeling [1], in dynamic analysis and simulation [9,10], in mapping techniques [2,6], etc.

The PCGs are annotated with appropriate information Parix needs for the creation and communication of its processes. PCGs are interpreted by a universal Parix Loader which initializes the applications specified by the PCGs. The annotated PCGs are produced from Ensemble scripts by two tools, the PCG-builder and the PCG-annotator.

2. The universal Loader acts as the “main” program for all Parix applications, which reads an annotated PCG and loads the appropriate processes on processors passing to them their communication and command line parameters. The programmer does not have to write a “main” program at all, only reusable components giving a result or providing a service.

3. The Loader requires that the program components from which processes are instantiated are reusable as library components. Program components do not assume any specific communication structure in which the processes instantiated from them are involved, but only generally specify the type of their communication channels. A reusable program component may have many process instantiations in the same application, each with its own communication dependencies. A reusable program component may also be used without modification in other applications, where its process instantiations communicate with altogether different processes, which are themselves instantiations of other reusable library components.

The structure of the paper is as follows: in section 2 we present the Process Communication Graphs, their annotation and their generation from Ensemble scripts; in section 3 we present the Parix Loader which interprets annotated PCGs; in section 4 we present the design of reusable program components in Parix; in section 5 we demonstrate the methodology by composing variations of Parix programs using the same reusable components and in section 6 we present our conclusions.

2. The PCGs and their annotation

Before we define the PCGs and their annotation let us describe a distributed application which we shall use as a demonstrating example.

2.1 A distributed application: Get Maximum

Processes instantiated from a terminal component possess a value; all terminal processes, or simply terminals, need to get the maximum value possessed by any of them. To limit the number of messages the terminals do not broadcast their values to all others; instead, there are processes, instantiated from a relay component, to which groups of terminals send their values. The relay processes, or simply relays, cooperate to find the maximum of the values, which they then send to their groups of terminals. The terminals have one communication dependency, that with their associated relay, which we call S (Server) type. The relays have two types of communication dependencies, one with their groups of terminals, which we call C (Client) type, and one with the relays, which we call P (Propagation) type. A relay may have any non negative number of C dependencies and P dependencies. The main actions and the communication dependencies of terminals and relays are:

The actions of a terminal	
send local value to relay	(to S type)
receive maximum value from relay	(from S type)

The actions of a relay	
receive values from the client terminals	(from C type)
find the local maximum (LM) of values	
send LM to all other relays	(to P type)
receive LMs from all other relays	(from P type)
find the global maximum GM	
send GM to its client terminals	(to C type)

The implementation should be easily configurable, that is, to be possible to add or remove terminal and/or relay processes, without any modification of the program components, i.e. the terminal and relay executables.

2.2 The elements of the PCG and their annotation

Processes will be depicted on PCGs by nodes comprised of two concentric circles depicted in Figure 1:

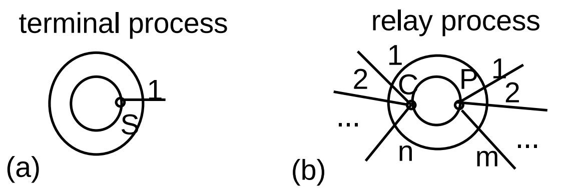

Figure 1: Graphical depiction of processes.

On the inner circle the type of dependencies are indicated. The inner circle depicts the general interface type of the program components. The arcs leaving the nodes indicate communication dependencies (of a specific type) with other processes. The points where the arcs cut the outer circle depict the actual interface of processes to other processes. Each point of intersection is called a *port* and is indexed by a unique positive integer within a port type. The arcs of the PCG connect ports of nodes. Let us assume, for example, that we have a configuration of eight terminals connected to four relays depicted in Figure 2.

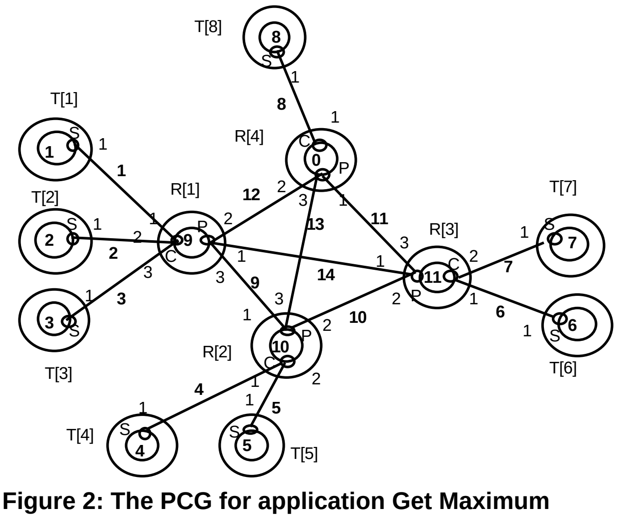

Figure 2: The PCG for application Get Maximum

The three C type ports of relay R[1] are connected with the S type ports of three terminals T[1], T[2] and T[3]; the two C type ports of relay R[2] are connected with the S type ports of two terminals T[4] and T[5]; the two C type ports of relay R[3] are connected with the S type ports of two terminals T[6] and T[7]; and finally the single C type port of R[4] is connected with the S type port of T[8]. All relays are connected to each other via their P ports. The elements of the PCG described so far specify a general PCG independent of any message passing environment.

For a PCG to specify a complete application it needs to be annotated. We need to specify on which processor each process will be loaded. Essentially, this is the mapping of the application as code is assigned to processors. The nodes are annotated by consecutive integers starting from zero, which are the processor identifiers in a Parix partition. In the PCG representation of Figure 2 the ProcIds appear at the centers of the circles.

Arcs on the PCGs represent communication channels. For a complete communication channel specification request identifiers are used by both sending and receiving threads. The request identifiers annotate the arcs of the PCG. Finally, nodes are annotated by the full path name of the executable component from which the process will be instantiated and possibly by command line parameters. For reasons of simplicity executable path names and parameters are not depicted on Figure 2.

The annotated PCG may be produced by a graphical tool or by a textual description. We have developed an Ensemble script language and script processing programs which read Ensemble scripts and produce annotated PCGs. A program script has three parts: the first describes the general PCG, the second the annotation of the PCG specific to a parallel environment (in this case Parix) and the third the annotation specific to the sequential components. Each of the three parts is further subdivided into sections.

The script generating the annotated PCG of Figure 2, is presented in Figure 3. The first part, headed by PCG, specifies the components of the application and their types of dependencies; then it defines the processes, instantiations of components, and their number of ports for each type. All T nodes have one port of type S, all R nodes have three ports of type P; node R[1] has three ports of type C, nodes R[2] and R[3] two ports of type C and R[4] one port of type C. This part also defines the channels between the ports. A script processing program, called PCG-builder, parses this first part of the script and produces the PCG of the application. It is the only tool common for all message passing environments.

The second part, headed by Parallel System defines the specific PCG annotation for Parix. The compulsory annotation of nodes by Process Identifiers, and of arcs by Request Identifiers is specified; here the default specifies the annotation of the nodes and arcs by unique non-negative integers, but other generating algorithms or direct annotations may be defined.

The third part, headed by Sequential System, annotates the nodes of the PCG with the file locations of the component executables from which processes are to be instantiated, as well as any command line parameters required by the processes. All T processes are passed an integer parameter which is the value they possess. The maximum is 999, the parameter of T[8]. A second script processing program, called PCG-annotator, parses the second and third parts of the script and produces the Parix annotated PCG of the application.

```
Application Get_Maximum;
PCG
Components
 T port-types:S;
 R port-types:C, P;
Processes
/* specify for each process the number of
ports of each type*/
 T[1], T[2], T[3], T[4],
 T[5], T[6], T[7], T[8] #ports =S:1;
 R[1]                   #ports = C:3, P:3;
 R[2], R[3]             #ports = C:2, P:3;
 R[4]                   #ports = C:1, P:3;
Channels
/* Connect process ports */
 T[1].S[1] <-> R[1].C[1];
 T[2].S[1] <-> R[1].C[2];
 T[3].S[1] <-> R[1].C[3];
 T[4].S[1] <-> R[2].C[1];
 T[5].S[1] <-> R[2].C[2];
 T[6].S[1] <-> R[3].C[1];
 T[7].S[1] <-> R[3].C[2];
 T[8].S[1] <-> R[4].C[1];
 R[1].P[1] <-> R[3].P[1];
 R[1].P[2] <-> R[4].P[2];
 R[1].P[3] <-> R[2].P[1];
 R[2].P[2] <-> R[3].P[2];
 R[2].P[3] <-> R[4].P[3];
 R[3].P[3] <-> R[4].P[1];
PARALLEL SYSTEM
Environment Parix;
 Parix_annotation
  ReqId: default; /* unique request Ids */
  ProcId: default; /* Processor Ids 0-N */
SEQUENTIAL COMPONENTS
Executable files
/* full path and name of executables*/
  R:"/home/users/bcast/relay";
  T:"/home/users/bcast/terminal;
Execution Parameters
T[1]:21; T[2]:22; T[3]:23; T[4]:24;
T[5]:25; T[6]:26; T[7]:27; T[8]:999;
```

Figure 3: The Ensemble script for Get Maximum

The annotated PCG depicts all the characteristics of the application: the number of processes of each component type and their interconnection; the processors to be loaded on; the request identifiers uniquely specifying the tags of messages. The annotated PCGs may now be interpreted by the Loader program, common for all Parix applications.

3. The Parix application loader

The Parix application Loader is the program that is responsible for launching the distributed application. It is the universal Parix “main” program which is loaded on all processors on the partition having a specific annotated PCG as its command line argument. Applications will be initialized with the run command issued from the Parix front-end in the form:

run -a “partition” Loader “annotated PCG”

The Loader is loaded on each of the processors in the partition. Its action is simple. Each copy first obtains the Processor Identifier it is running on, say MyProcId. It then visits the annotated PCG's nodes and, if a node is allocated to MyProcId, it loads the process denoted on the node passing it the appropriate connectivity and command line parameters. The connectivity parameters are a list of values providing the connectivity information for each of its ports. For each port two values must be provided (ProcId, ReqId). For example when creating R[2] the port connectivity information passed as parameters is

connectivity parameters	**port connectivity explanation**
C, 1, 4, 4	port C1 connected to Proc 4; ReqId 4
C, 2, 5, 5	port C2 connected to Proc 5; ReqId 5
S, 1 , 9, 9	port S1 connected to Proc 9; ReqId 9
S, 2, 11, 10	port S2 connected to Proc 11;ReqId 10
S, 3, 0, 13	port S3 connected to Proc 0; ReqId 13

The Loader calls the Parix Execute routine, having as first parameter the name of the executable file to be loaded and as a second its connectivity parameter list, followed by the command line parameter list. The processes are now appropriately loaded. They have now to be interconnected as specified by the parameter list. This is the responsibility of the reusable components.

4. The reusable program components

Reusability of program components demands that, if required by an application, any number of processes can be instantiated on any processor. It should also be possible to dynamically establish upon process creation the appropriate communication channels between them. As the number of actual processes and their communication dependencies is not fixed, the program components should only specify the number and type of communication channels in a general way. A program component should provide the means for the establishment of actual communication channels between any process created from it with any other compatible process without relying on their position.

A general channel specification in Parix is defined as a data structure, called port, storing (Processor Identifier, Request Identifier) pairs, (ProcId, ReqId) for short. The two elements in the port structure (ProcId; ReqId) is the primary connectivity information of a communication channel associated with a port. The actual port values are provided, at process creation time, by the Loader. A program component may manage many ports of the same type, which are organized as an array of ports of the this type. Furthermore, a component may have many types of ports. All ports of all its types form the component's interface and all port values are organized in the structure Interface. Each port in Interface is now identified by its index type and its port number within the type. Without loss of generality, all random communication routines (e.g., SR: SendNode, RecvNode; AR: PutMessage, GetMessage) and virtual link creation routines (e.g., ConnectLink) have to use as their port information parameters elements of the Interface structure: Interface[T].port[P].ProcId and Interface[T].port[P].ReqId, where T is a port type and P a port number. Structure Interface may have for each port two more elements: one of type Link holding the link value of a channel which link communication routines (e.g., SL: SendLink, RecvLink) may use as their parameter the value (Interface[T].port[P].Link) and one of type topology holding the topology value of a link, which topology communication routines (e.g., ST: Send, Recv; AT: ASend, ARecv) may use as their parameter the value (Interface[T].port[P].Top).

We permit components to have a variable number of ports of each type; the interface for each process, the number as well as the values of its ports, will be fixed upon its creation. The loader provides this information in the parameters of Execute calls for process creation; the port interface of the process on processor 10, i.e. R[2], for example, will have the following values:

port	ProcId	ReqId
C.1	4	4
C.2	5	5
S.1	9	9
S.2	11	10
S.3	0	13

It is the responsibility of the components to read their parameters and fix the shape and values of elements of structure Interface upon process creation. All components have a common structure, shown in Figure 4:

```
void main(argc, argv);
  InterfaceType Interface[N]; /* N is
#types*/
{ MakePorts(Interface);
  SetInterface(Interface);
  realMain(Interface); /* main action */ }
void realMain (Interface, argc1, argv1);
     { actions of components}
```

Figure 4: The common structure of components

The programmer has to fix for each component the number N of its port types. As each process is created, it calls the MakePorts routine which sets the appropriate number of ports of each type. Then SetInterface routine is called which sets the (ProcId, ReqId) values in the appropriate ports of structure Interface. Having set up its port interface routine realMain is called which codes the main actions of a component. The complete application is now running.

In the description of routine SetInterface in the previous paragraph, it is assumed that realMain uses random communication, as in Interface only the primary connectivity information is stored. The Link and the Topology fields have not been fixed. If we simply continue the actions of SetInterface by a series of ConnectLink calls, the components may not be reusable at all; as a matter of fact the program may not be able to run at all, coming to a deadlock. The reason is that for the establishment of virtual links, symmetric synchronous calls have to be made by the participating threads. Each thread calls the ConnectLink routine, which in order to create the link requires synchronous communication between the threads. This means that both threads should be able to reach their corresponding ConnectLink calls. The general problem is exemplified in the Parix manual [8], chapter 10, pg. 90-97, where a ring topology is created. There, N processes are to be connected in a ring; each has two links, one left and one right. If all processes try first to connect to their left processor and then to their right, a deadlock occurs, as all of them wait for the left process to issue a corresponding call. The solution suggested is for the processes residing on a even ProcId to connect to their right first and then to their left, and for processes residing on an odd ProcId to connect first to their left and then to their right. This example demonstrates that for the simplest of topologies care should be given in the order of calling ConnectLink routines. Significant design effort and programming is required for more complex topologies. For program components to be completely reusable as library components they cannot depend on any ConnectLink calls ordering. For otherwise, components would depend on each other, contradicting their reusability as library components. Therefore, the alternative suggested in [8] is followed, by which each ConnectLink call is issued by a different thread; for each communication channel one thread is needed, the activation of which is performed by Parix. No extra effort is required by the programmer.

The alternative method is coded in the SetInterface routine. Rather than having a number of ConnectLink calls, a number of StartThread calls are issued. Each StartThread invokes the program SetLink which takes as parameter a port and calls ConnectLink with parameters the port values ProcId and ReqId. The ConnectLink call sets the Link field of the port. If in addition this link is part of a topology, program SetLink sets the topology field of the port. All SetLink threads are synchronized before SetInterface exits. Then realMain may use any type of communication permitted by Parix.

Components are now completely reusable as they only specify general methods for setting their communication interface, they do not assume communication with any specific components and the virtual links between processes are established in any application context by asynchronous threads. Furthermore, components do not specify on which processor processes should be loaded. This information is specified in the script.

Running the Parix Loader with the annotated PCG of Get Maximum as input we get the following output; the first part is produced by the Loader, as it spawns processes, and the second by the terminal processes:

```
Spawn process 1 (terminal) on Proc 1
Spawn process 2 (terminal) on Proc 2
Spawn process 3 (terminal) on Proc 3
Spawn process 4 (terminal) on Proc 4
Spawn process 5 (terminal) on Proc 5
Spawn process 6 (terminal) on Proc 6
Spawn process 7 (terminal) on Proc 7
Spawn process 8 (terminal) on Proc 8
Spawn process 9 (relay) on Proc 9
Spawn process 10 (relay) on Proc 10
Spawn process 11 (relay) on Proc 11
Spawn process 12 (relay) on Proc 0
```

```
[Proc 1] The maximum value is 999
[Proc 2] The maximum value is 999
[Proc 3] The maximum value is 999
[Proc 8] The maximum value is 999
[Proc 4] The maximum value is 999
[Proc 6] The maximum value is 999
[Proc 5] The maximum value is 999
[Proc 7] The maximum value is 999
```

As the twelve processes, eight terminal and four relay are spawned, the Parix Loader prints the processor they run on; the terminal processes print the global maximum of their parameters. All terminal processes print the same maximum of 999 which was the parameter of T[8].

For a Parix program to behave correctly, the nodes on the PCG and the actual program components must be compatible, that is, they should specify, the former virtually and the latter actually, the same number of types of ports. Furthermore, the connections between ports should be of compatible type, that is they agree on the type of messages they exchange and their management. The present version of the Loader does not check the compatibility of the connections. We are currently investigating formal methods for describing and testing compatibility, which will be integrated in the Loader.

Having developed reusable components we may use them in scripts to compose new applications. The script language is flexible and permits the rapid composition of Parix programs. It is straight forward to edit scripts to scale a program by adding and connecting new components, to change the allocation of processes to processors, to change the topology of the components, etc., without modifying the program components. In the next section we demonstrate the flexibility of the script language to compose applications by reusing components in different topologies.

5 Variations of Get Maximum

The specification for the Get Maximum program in section 2 did not specify any particular topology by which the relay processes should be connected. In the solution of section 2 we had adopted a topology in which all relay processes are connected with each other. We may achieve the required functionality by adopting different topologies. We shall present two variations, one in which relay processes form a star topology and a second in which they form a tree topology. For these variations we will modify the scripts and reuse the terminal and relay components.

5.1 Get Maximum by star topology

In this solution we use an extra relay process to which the old four relay processes will be connected. The four relay processes have now only one P port, through which they send the maximum value received from their terminals. The new relay process, let us call it central, has four ports of type C (clients). The P type ports of the four relay processes are connected to the C type ports of the central process! Let us note, that the C and the P ports of the relay processes are compatible, as only one value is sent and one value is received through them. The PCG for this configuration is depicted in Figure 5:

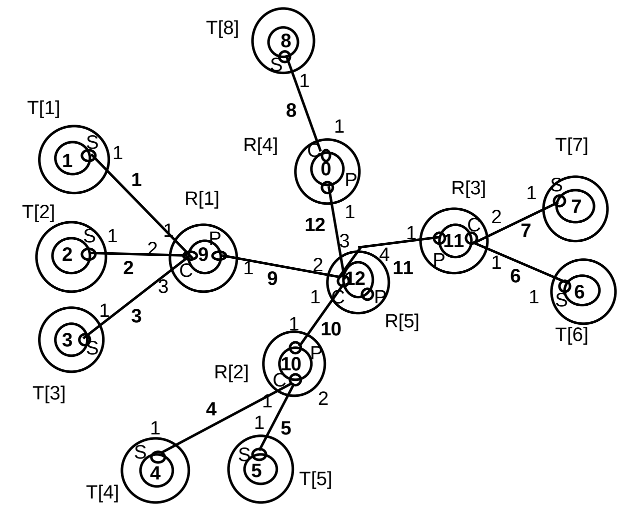

Figure 5: The PCG of Get Maximum by Star

Let us describe the behavior of the program: the four relay processes, as before, select the local maximum of the values of their clients and propagate it via their single port of type P to the central relay process. The central process receives values from its C ports and selects their maximum. There are no P ports to send the maximum. It then sends its maximum to its C ports. What actually sends is the global maximum, as it is the maximum of all maxima. Each relay receives the global maximum, but, according to the algorithm, they act as if it is the local maximum of a relay process. They compare it with their own maximum, select the value they have received and send it to their client ports. The PCG part of the modified script is in Figure 6:

```
Application Get-Maximum-Star;
PCG
Components
 T port-types:S;
 R port-types: C, P;
Processes
 T[1], T[2], T[3], T[4],
 T[5], T[6], T[7], T[8] #ports = S:1;
 R[1]                   #ports = C:3, P:1;
 R[2], R[3]             #ports = C:2, P:1;
 R[4]                   #ports = C:1, P:1;
 R[5]                   #ports = C:4, P:0;
Channels
   T[1].S[1] <-> R[1].P[1];
   T[2].S[1] <-> R[1].P[2];
   T[3].S[1] <-> R[1].P[3];
   T[4].S[1] <-> R[2].P[1];
   T[5].S[1] <-> R[2].P[2];
   T[6].S[1] <-> R[3].P[1];
   T[7].S[1] <-> R[3].P[2];
   T[8].S[1] <-> R[4].P[1];
   R[1].P[1] <-> R[5].S[1];
   R[2].P[1] <-> R[5].S[2];
   R[3].P[1] <-> R[5].S[3];
   R[4].P[1] <-> R[5].S[4];
```

Figure 6 The PCG for Get Maximum by Star

For this solution no changes are needed for the terminal and the relay program components, but only to the script. The executables of the terminal and relay program component were reused. The new program script was produced rapidly by modifying the program script of the version of section 2. The annotated PCG was produced from the script, which was given as input to the Loader.

5.2 Get Maximum by tree topology

In this variation we maintain the relationship of the eight terminals to the four relay processes having, as in the star solution, one P port. The P ports of R[1] and R[2] are connected with the C ports of R[5] and the P ports R[3] and R[4] are connected with the C ports of R[6]. Both R[5] and R[6] have two C ports and one P port; their P ports are connected to the two C ports of R[7], which does not have any P ports. The process structure is a tree of height 3: the terminal processes are the leafs; R[1], R[2], R[3] and R[4] at level two; R[5] and R[6] at level one; and R[7] as the root. At each level, the relay processes receive the values from their clients, select the maximum and propagate it to the next level up. The root selects the maximum and sends it to its client processes. The relay processes below the root do the same until the maximum reaches the terminal processes. The PCG part of the script is shown in Figure 7:

```
Application Get-Maximum-Tree;
PCG
Components
 T port-types:S;
 R port-types: C, P;
Processes
 T[1], T[2], T[3], T[4],
 T[5], T[6], T[7], T[8] #ports= S:1;
 R[1]                   #ports= C:3,P:1;
 R[2], R[3], R[5], R[6] #ports= C:2,P:1;
 R[4]                   #ports= C:1,P:1;
 R[7]                   #ports= C:2,P:0;
Channels
   T[1].S[1] <-> R[1].C[1];
   T[2].S[1] <-> R[1].C[2];
   T[3].S[1] <-> R[1].C[3];
   T[4].S[1] <-> R[2].C[1];
   T[5].S[1] <-> R[2].C[2];
   T[6].S[1] <-> R[3].C[1];
   T[7].S[1] <-> R[3].C[2];
   T[8].S[1] <-> R[4].C[1];
   R[1].P[1] <-> R[5].C[1];
   R[2].P[1] <-> R[5].C[2];
   R[3].P[1] <-> R[6].C[1];
   R[4].P[1] <-> R[6].C[2];
   R[7].C[1] <-> R[5].P[1];
   R[7].C[2] <-> R[6].P[1];
```

Figure 7: The PCG for Get Maximum by Tree

The PCG part of the script is close to the design of the message passing application, the other two parts are more closely to the detailed design and the implementation of the application.

6. Conclusions and future work

The Ensemble methodology applied to Parix is presented. Ensemble is a message passing program implementation methodology by which the programming overhead effort required for creating arbitrary structured parallel applications in Parix is overcome. In Ensemble parallel applications are virtually specified by Process Communication Graphs (PCGs) interpreted by a universal Parix Loader, which acts as a universal Parix “main” program. The loader automatically creates the appropriate processes establishing their communication dependencies. Program component structures have been developed permitting their reusability as library components.

The Ensemble methodology “removes” from message passing environments the aspects that most influence the architecture of applications, namely process management. They are “removed” in the sense that it is not required for program components to define topologies, the processors they are running on, etc. All process management aspects are expressed in the scripts, which are virtual representations of applications. Consequently, Ensemble “removes” the effort of programming most of the architectural idiosyncrasies of message passing environments.

Ensemble does not suggest a new message passing environment and does not demand any changes to message passing environments 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 minimal compared with the saved programmer’s development time. Ensemble is independent of any design, visualization, performance, etc. tools.

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 Parix programs. It is straight forward to edit the script to scale a program by adding and connecting new components, to change the allocation of processes to processors, the topology, etc.

Ensemble separates all elements of a message passing application: the process topology, the architecture, the mapping of the processes onto processors and the activity of the application giving the required result or providing a 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.

Ensemble defines reusable message passing library components with scalable communication interfaces and uses them in the composition of applications in a “soft LOGO” manner.

We demonstrated the flexibility of the methodology, by composing various solutions to the Get Maximum problem. Having constructed the program components for the first solution we used them to compose and execute other Parix programs solving the same problem but by a different design.

The methodology may be applied to other message passing environments by developing suitable PCG annotation principles, structure of reusable components and Loaders. We have applied it for PVM [5]. Although PVM imposes an altogether different process management model than Parix based, for example, on parent-child creation of processes, it was possible to apply, in principle, the same methodology. Ensemble implementations of the same design look similar and components identical. Also, the portability of applications from one environment to the other is possible and in some cases is done automatically. We shall compare implementations of the methodology under PVM and Parix in a future report.

Future plans include the development of Ensemble for MPI, parametric topology descriptions, more type of process interactions. Ensemble is related to the composition of Object Oriented systems by using objects and scripts [7] encouraging a component oriented approach to application development. The PCGs may be viewed as the scripts that bind components together. We shall pursue this comparison further in the future.

References

[1] G.R.Andrews: ‘Paradigms for Process Interaction in Distributed Programs’, ACM Computing Surveys, Vol. 23, No.1, March 91.

[2] F. Berman, L.Snyder, ‘On mapping parallel algorithms into parallel architectures’, J. Parall. Distrib. Comput. 4, 5, 439-458.

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

[4] J.Y.Cotronis: Efficient composition and automatic initialization of arbitrary structured PVM programs, IFIP Proceedings of the 1st Workshop on Software Engineering for Parallel and Distributed Systems, ICSE 96, Berlin, March 96.

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

[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] Parix1.2, Manual.

[9] 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.

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