In Proceedings Recent Advances in PVM and MPI, V. Alexandrov and J. Dongarra (eds.), LNCS 1497, 1998, pp. 145-152

Developing Message–Passing Applications on MPICH under Ensemble

Yiannis Cotronis

Dep. of Informatics, Univ. of Athens, Panepistimiopolis, 17184 Athens, Greece
Phone:+301 7275223, Fax:+301 7219561, e–mail:cotronis@di.uoa.gr

Abstract. We present an implementation methodology for message–passing applications, called Ensemble applied to MPI. Applications are implemented by reusable executables having open, scalable interfaces and by scripts specifying process communication graphs, annotated with application execution information. An MPICH procgroup file is generated from scripts and executed my mpirun.

1 Introduction

The design and implementation of message–passing (MP) applications have been recognized as complex tasks. Designing MP applications involves designing the combined behavior of its processes, that is, their individual execution restricted by their communication and synchronization interactions. Implementation of MP applications involves programming of its sequential processes, as well as (latent) management of processes and architecture resources. The emergence of Message Passing Environments (MPE), such as PVM [5] and MPI [7], or more correctly its implementations (e.g. [2, 1]), provide abstractions of architectures significantly simplifying process and resource management.

Nevertheless, the step from design to implementation remains, in general, a demanding task, requiring special effort. The main implementation complexity arises from the process management models of MPEs. Each MPE is suited for specific types of process topologies, those being closer to its process management model. For example, PVM and, respectively MPI favor tree and, respectively ring or grid topologies. Topologies not well suited to an MPE may certainly be implemented, but require complex programming. Consequently, the implementation effort does not depend only on the design, but also on the target MPE, each requiring different techniques. More effort is required for implementing scaling of topologies and code reuse.

We have developed an implementation methodology, called Ensemble, alleviating the above problems. Implementations maintain the original design, look similar on different MPEs, are easily maintainable, scalable, reusable and mechanically portable to other MPEs. In section 2, we examine implementation complexities under MPICH; in section 3, we outline Ensemble and tools for MPICH; in 4, we demonstrate the reusability of executables; and in 5, we present our conclusions.

2 Implementation on MPI and MPICH on Clusters

Process management, in general, involves process creation and their identification, and establishment of communication channels. In MPICH, processes are created by the `mpirun` command. If the implementation is organized in a SPMD style the command `mpirun -np<np><progname>` runs `progname` on `np` processors. If an application is organized in an MPMD style, a procgroup file is needed, which contains a list of executables in the form `<hostname> <#procs> <progname>`. The command `mpirun -p4pg<procgroup><myprog>` starts the application, where `myprog` is started directly. Mpirun has a number of other options, which are beyond the scope of exploring here. In all cases, mpirun manages processes and architecture resources.

Processes in MPI are identified by their `rank` number. Similarly to other MPEs, channels in MPI are established implicitly by providing the appropriate process identifiers (rank), tagging identifiers (tagid) and context (communicator), as parameters to communication routines. Such implicit channel establishment demands programming and its complexity depends on the topology, as well as on the MPE.

In a regular topology, rank identifiers may be associated to process positions by functions and, consequently, each process may determine its communicating processes from its own rank. Scaling regular topologies is implemented by parameterizing process position functions. Designs of irregular process topologies are much more difficult to implement, as the association of process ranks with their position is not possible, in general. Scaling of irregular topologies is also difficult to implement, as there may only be local, regular sub-topologies. Finally, reusability of executables is limited as they are programmed to operate in a fixed topology.

To alleviate such problems we propose a scheme for establishing channels explicitly, rather than implicitly. We design executables with open communication interfaces, a set of communication ports. Ports in MPI are structures of triplets (rank, tagid, and communicator), which specify communication channels. Communication routines in programs refer to such (open) ports, rather than using triplets directly. Processes get values for their ports upon their creation.

Topology creation and scaling requires the replication of processes and establishing channels, by symmetric and consistent port assignments. For topologies, such as a torus or a ring, it is sufficient to replicate identical processes, each having the same number of ports. However, for other topologies, such as master/slave or client/server, each process may have any number of ports, possibly within a range. We thus permit scaling of interface ports for individual processes. In this scheme (process creation, interface scaling, interface assignments) any topology, regular or irregular, may be easily established and scaled. Furthermore, programs are reusable like library components, as they do not rely on communicating with any specific processes. In the next section we describe the Ensemble tools which support this scheme.

3 MPICH and the Ensemble Methodology and Tools

Ensemble implementations consist of an 'ensemble' of: 1. Scripts specifying annotated Process Communication Graphs (PCG). PCGs depict the process topologies; nodes and arcs of PCGs are annotated with MPICH related information required for process creation on specific architecture and communication. 2. The reusable executables, which are message–passing library components having open scalable interfaces. 3. The Loader, which interprets the annotated PCGs and runs applications. We simply use mpirun on an appropriate procgroup file generated from the annotated PCG.

We demonstrate Ensemble by implementing the application Get Maximum. The requirement is simple: there are Terminal processes, which get an integer parameter and require the maximum of these integers. We design the following solution, called Terminal–Relays–in–Ring: Terminals are connected as client processes to associated Relay processes. Each Terminal sends (via port Out) its integer parameter to its Relay and, eventually, receives (via port In) the required maximum. Relays receive (via their Cin ports) integer values from their client Terminals and find the local maximum. Relays are connected in a ring. They find the global maximum by sending (via Pout port) their current maximum to their next neighbor in the ring, receiving (via Pin port) a value from their previous neighbor, comparing and selecting the maximum of these two values. Relays repeat the send–receive–select cycle M-1 times, where M is the size of the ring. Finally, Relays send (via Cout ports) the global maximum to their client Terminal processes. The topology or process communication graph (PCG) of the design is depicted in Fig. 1. The design is irregular, as Relays are associated with distinct numbers of Terminals.

3.1 The Application Script

The script is directly obtained from the design and abstractly specifies an implementation: the program components; the processes and their interface; the communication channels; the application parameters; the use of architecture resources. The script for Terminal-Relays-in-Ring for three Relays and six Terminals is depicted in the first column of Fig. 1.

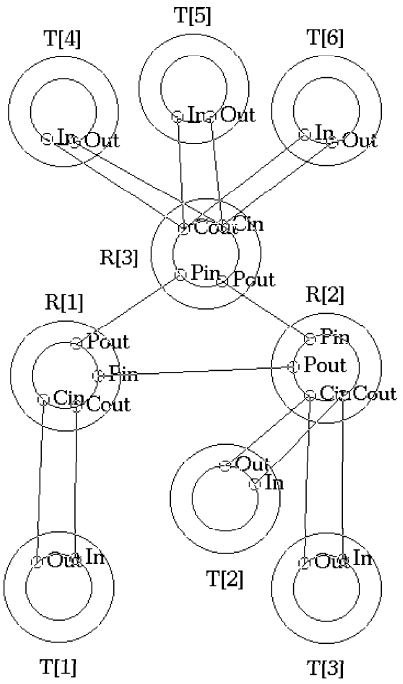
The script is structured in three main parts. The first part, headed by `PCG`, specifies the PCG of the application, which is independent of any MPE or architecture. The PCG part has three sections. In the `Components` section, we specify abstractions of the components involved in the topology (e.g. Terminal) by their name (e.g. T), their communication interface and their design parameters, explained in the sequel. For components we specify their communication port types and the valid range of ports of each type in brackets. Communication types are depicted on PCGs on the inner circle of process nodes. The actual ports are depicted on the outer circle. The Terminal component, for example, has two synchronous communication types (In and Out), whilst the Relay component has four, two asynchronous (Cin, Cout) and two synchronous (Pin, Pout). Types In and Out of Terminal have exactly one port. A Relay process may have

Application Script

```
APPLICATION Get_Maximum_by_Ring;
PCG
Components
  T:In,Out(Synch)[1..1]
  R:Pin,Pout(Asynch)[0..1],
    Cin,Cout(Synch)[0..];
    design: M;
Processes
  T[1],T[2],T[3],
  T[4],T[5],T[6]#In,Out:1;
  R[1]#Cin,Cout:1,Pout,Pin:1;M=3;
  R[2]#Cin,Cout:2,Pout,Pin:1;M=3;
  R[3]#Cin,Cout:3,Pout,Pin:1;M=3;
Channels
  T[1].Out[1] -> R[1].Cin[1];
  T[2].Out[1] -> R[2].Cin[1];
  T[3].Out[1] -> R[2].Cin[2];
  T[4].Out[1] -> R[3].Cin[1];
  T[5].Out[1] -> R[3].Cin[2];
  T[6].Out[1] -> R[3].Cin[3];
  T[1].In[1] <- R[1].Cout[1];
  T[2].In[1] <- R[2].Cout[1];
  T[3].In[1] <- R[2].Cout[2];
  T[4].In[1] <- R[3].Cout[1];
  T[5].In[1] <- R[3].Cout[2];
  T[6].In[1] <- R[3].Cout[3];
  R[1].Pout[1] -> R[2].Pin[1];
  R[2].Pout[1] -> R[3].Pin[1];
  R[3].Pout[1] -> R[1].Pin[1];
```

PCG Builder

PCG and its Annotation




```
PARALLEL SYSTEM
MPI on cluster;
  tagID : default;
  communicators : default;
  com_functions : default;
Process Allocation
  T[1],R[1] at zeus;
  T[2],T[3],R[2] at gaia;
  T[4],T[5],T[6],R[3] at chaos;
Executable Components
  T : path default file Trm.sun4;
  R : path default file Rel.sun4;
APPLICATION PARAMETERS
  T[1]:"6";T[2]:"999";T[3]:"7";
  T[4]:"8";T[5]:"9";  T[6]:"5";
```

PCG Annotator

```
Node 1   name:T[1]
   allocation:zeus
   file      :Trm.sun4
   path      :default
   parameters:6
        ...
Node 5   name:R[2]
   allocation:gaia
   file      :Rel.sun4
   path      :default
   parameters:3
         ...
Channel 1(A):1.Out->2.Cin tag 1
Channel 4(A):6.Out->9.Cin tag 4
Channel 5(A):7.Out->9.Cin tag 5
         ...
```

Fig. 1. The application script and the annotated PCG of Terminal-Relays-in-Ring

any number of Terminals as its client processes. Types Pin and Pout may have at most one port (there are no ports when there is only one Relay). Types Cin and Cout have any positive number of ports.

We also specify parameters, which are required for the correct behavior of the executables and are related to the designed topology and not to the application requirement. For this reason, they are called design parameters. For example, Relays must repeat the send–receive–select cycle M-1 times. The value M depends on the size of the ring and is crucial for producing the correct result.

The `Processes` section specifies the nodes of the PCG, naming them by uniquely indexing component names (e.g. T[1],...,T[6] and R[1],R[2],R[3]) setting their number of ports for each communication type and values for their design parameters. All Terminal nodes have exactly one port of each type. Each of the three Relay processes has a distinct number of ports of types Cin (respectively Cout), depending on the number of clients they have, and one port of types Pin (respectively Pout). The design parameter of all Relays is set to 3, the size of the Relays ring. The `Channels` section specifies point–to–point communication channels, by associating ports. A tool program, the PCG–builder, reads the PCG part and actually generates a representation of the PCG, giving rank identifiers to the nodes.

The second part of the script, headed by `PARALLEL SYSTEM`, specifies information for processes and channels required by MPICH running on clusters. It specifies for each channel the values for tagid and communicator; In the example script `default` indicates unique tagid generation and MPI_COMM_WORLD, respectively. The rank is obtained from the rank of the node at the other end of arcs. In addition we may specify the type of communication to be used (blocking, standard, etc.) when using a port. The allocation of processes is also specified, followed by the file names of executables. Finally, the third part of the script, headed by APPLICATION PARAMETERS, specifies process parameters required in the application. In the example script, each Terminal is given an integer parameter. The second and third parts are interpreted by the PCG–annotator, appropriately annotating nodes and arcs of the PCG created by the PCG–builder. In Fig. 1 below the general PCG, the annotation of some of its nodes and channels for MPICH is shown.

3.2 The Reusable Executables

The Ensemble components compute results and do not involve any process management or assume any topology in which they operate. They have open ports for point-to point communication with any compatible port of any process in any application and are reusable as executable library components. A port in MPI is implemented as a structure, which stores a triplet of (rank, tagid, communicator). Ports of the same type form arrays and arrays of all types form the interface of the component. All send and receive operations in processes refer to ports, identified by a communication type and a port index within the type. As processes are created, they get the actual number of ports of each type from their command line parameters. Processes scale their interface by executing a routine,

called MakePorts. Processes also get the values for the communication parameters for each port from their command line parameters. Processes set actual values to their interface by executing a routine, called SetInterface. Each MPE demands its own MakePorts and SetInterface routines. A skeleton for program components, which may be seen in Fig. 2 has been developed. In association with wrapper routines Usend and Ureceive, which call MPI routines, all differences are hidden and the appearance of components of any MPE is unified.

```
/* Terminal Code */
void RealMain(Interface,argc,argv)
   struct port_types *Interface;
   int argc; char **argv;
{ int Typecount=2;
  GetParams(V);
  Usend(Out[1],V);
  Ureceive(In[1],Max);}
-----------------------------------
/*Common main for all executables*/
void main(argc,argv)
     int argc; char **argc;
{ struct Ps
       {int rank,tag,comm,Ftype;}
  struct Pt
       {int PCount;struct Ps *port;}
  typedef struct Pt InterfaceType;
  extern int TypeCount;
  InterfaceType Interface;

  MakePorts(Interface,TypeCount);
  SetInterface(Interface,TypeCount);
  RealMain(Interface,argc,argv); }
```

```
/* Relay Code */
void RealMain(Interface,argc,argv)
     struct port_types *Interface;
     int argc; char **argv;
{ int Typecount=4;
  GetParam(M);
  LMax=0;
  for(j=1;
      j<=Interface[Cin].PCount;
      j++)
  { Ureceive(Cin[j],V);
    if (V > LMax) Lmax=V;}

  Gmax=Lmax;
  for (i=1;i<=M-1;i++)
  {  Usend(Pout[1],GMax);
     Ureceive(Pin[1],V);
     if(V > GMax)Gmax=V;}

  for(j=1;
      j<=Interface[Cout].PCount;
      j++)
  USend(Cout[j],Gmax);}
```

Fig. 2. Terminal and Relay program components

For each component, we set the number of communication types that it requires (TypeCount), as specified in the design; Terminal has two and Relay four communication types. Processes first call MakePorts to set–up the appropriate number of ports of their Interface, then call SetInteface to set values to ports of Interface and they call their RealMain actions. Using the component skeleton, only RealMain, the actions related to the application, needs to be coded.

3.3 Executing Applications

A pre–loader program visits the PCG nodes and generates the `procgroup` file of executables and the appropriate parameters, which `mpirun` uses to create processes. All necessary information annotates the PCG: the process rank, the

associated executable and its host allocation; its design and application parameters and the number of ports and of each type as well as their values.

4 Design and Implementation Variations

In this section, we demonstrate the reusability of executables. We design a variation of Get Maximum, which reuses Terminal and Relay. The application script and the PCG of the application are depicted in Fig. 3.

```
Components
 T:In,Out(Asynch)[1..1];
 R:Pin,Pout(Asynch)[0..1],
   Cin,Cout(Asynch)[0..];design:M;
Processes
  T[1],T[2],T[3],
  T[4],T[5],T[6] #C:1;
  R[1]#Cin,Cout:2,Pout,Pin:1;M=2;
  R[2]#Cin,Cout:3,Pout,Pin:1;M=2;
  R[3]#Cin,Cout:3,Pout,Pin:0;M=1;
Channels
  T[1].Out[1] -> R[1].Cin[1];
  T[2].Out[1] -> R[2].Cin[1];
  T[3].Out[1] -> R[2].Cin[2];
  T[4].Out[1] -> R[3].Cin[1];
  T[5].Out[1] -> R[3].Cin[2];
  T[6].Out[1] -> R[3].Cin[3];
  T[1].In[1] <- R[1].Cout[1];
  T[2].In[1] <- R[2].Cout[1];
  T[3].In[1] <- R[2].Cout[2];
  T[4].In[1] <- R[3].Cout[1];
  T[5].In[1] <- R[3].Cout[2];
  T[6].In[1] <- R[3].Cout[3];
  R[3].Cout[2] -> R[1].Pin[1];
  R[3].Cout[3] -> R[2].Pin[1]
  R[1].Pout[1] -> R[3].Cin[2];
  R[2].Pout[1] -> R[3].Cin[3];
```

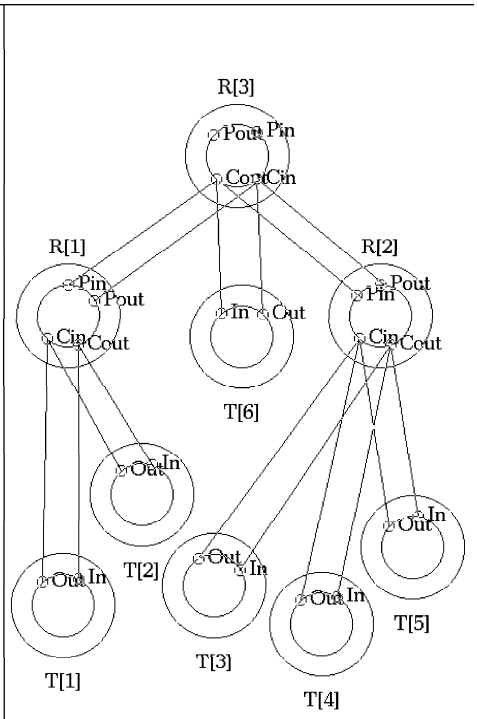


Fig. 3. The application script and PCG for Get Maximum Selectors and Relays in tree

In this variation, Relay processes are organized in a tree, with R[3] being the root, which has no Pout and Pin ports. R[1] and R[2] have one Pout and one Pin ports, which are connected to the Cin and Cout ports, respectively, of their parent R[3]. R[3] has also T[6] connected as a client process. R[1] has T[1] and T[2] as its client processes and Server[2] has T[3], T[4] and T[5] as its client processes.

We have specified that all communication is asynchronous for the ports to be compatible for communication. R[1] and R[2] receive values from their clients,

find their local maximum and send it to their Pout port, which is connected to a Cin port of R[3]. R[3] finds the global maximum, and as it is not connected in a ring, it directly sends the global maximum to its clients. T[6] gets the global maximum, as well as R[1] and R[2]. R[1] and R[2] "select" the global maximum and send it to their client processes.

5 Conclusions

We have presented Ensemble and its tools applied to MPICH. Ensemble structures implementations into scripts, which specify process topologies, channel specifications and resource allocation, and into program components, which encode application computations. As the elements of the implementations are separated, they may be modified independently of one another, thus simplifying program debugging and maintenance.

Portability does not necessarily imply that the same source runs efficiently on any architecture. For a ported program to run efficiently it usually has to be adapted and fine–tuned. Ensemble permits rapid program variations.

We have applied Ensemble [4] [3] on PVM and on Parix [8], respectively. Although, MPEs differ [6] Ensemble implementations look similar in MPI, PVM and Parix and porting them from one environment to the other is mechanical. Ensemble does not suggest a new MPE, it does not demand any changes to MPEs and its tools are independent of any design, visualization, performance, or any other tools which may be used with an MPE. Future work includes high level parametric topology descriptions in scripts, support for dynamic process creation, tools for porting applications to different MPEs.

References

1. Alasdair, R., Bruce, A., Mills, J.G., Smith, A.G.: CHIMP/MPI User Guide. EPCC, The University of Edinburgh
2. Bridges, P., Doss, N., Gropp, W., Karrels, E., Lusk, E., Skjellum, A.: User's Guide to MPICH, a Portable Implementation of MPI. Argone National Laboratory, 1995.
3. Cotronis, J.Y.: Efficient Program Composition on Parix by the Ensemble Methodology. In: Milligan, P., Kuchcinski, K. (eds), 22nd Euromicro Conference 96, IEEE Computer Society Press, Los Alamitos, 1996, 545–552
4. Cotronis, J.Y.: Message–Passing Program Development by Ensemble. In: Bubak, M., Dongarra, J. (eds.): Recent Advances in PVM and MPI. LNCS Vol. 1332, Springer–Verlag, Berlin Heidelberg New York (1997) 242–249
5. Geist, A. , Beguelin, A. , Dongarra, J. , Jiang, W. , Manchek, R. , Sunderam, V.: PVM 3 User's guide and Reference Manual. ORNL/TM–12187, May 1994
6. Gropp, W., Lusk, E.: Why Are PVM and MPI So Differrent? In: Bubak, M., Dongarra, J. (eds.): Recent Advances in PVM and MPI. LNCS Vol. 1332, Springer–Verlag, Berlin Heidelberg New York (1997) 3–10
7. M.P.I. Forum: MPI: A Message–Passing Interface Standard. International Journal of Supercomputer Applications, 8(3/4),1994
8. Parsytec Computer Gmbh.: Report Parix 1.2 and 1.9 Software Documentation. 1993