Interconnecting Multiple Heterogeneous Parallel Application Components

Pedro D. Medeiros   José C. Cunha

Universidade Nova de Lisboa – Faculdade de Ciências e Tecnologia
Departamento de Informática
Portugal
e-mail: {pm,jcc}@di.fct.unl.pt

Abstract. We present an infrastructure for building parallel applications by interconnecting slightly modified pre-existing parallel components. This infrastructure (called PHIS) allows the cooperation of components that run in different parallel machines. In succession, we describe the rationale behind PHIS, the primitives used to interconnect the application components and its internal architecture and we compare PHIS to related systems. Finally, we present an application where PHIS is used to interconnect several distinct components that define a parallel heterogeneous computational steering architecture for genetic algorithm applications.

1 Introduction

The work described here is integrated in an ongoing project having the main goal of developing an execution environment targeted at a heterogeneous set of machines, connected by a local area network, including distributed-memory multiprocessors. Our interest by heterogeneous architectures was motivated by some of the applications that we use as testbeds. In fact, some of these applications (for example computational fluid dynamics simulation and application of genetic algorithms to environmental science problems) are suited to a decomposition where the following components can be found:

- One or more computationally intensive components support the parallel execution of algorithms that simulate some physical process, using some parameters that can be modified during the execution. In some cases, this simulation could be itself decomposed into several components where each one could benefit from a different programming model which, in turn, could be more adapted to a particular hardware platform.
- A visualization component supports on-line data visualization.
- An interactive control component allows modification and inspection of the parameters of the simulation(s).

In these kind of environments, it should be possible to develop each component independently from the others and to reuse already existent software as much as possible.

1.1 Requirements for a interconnection system

In order to support the configuration and the data exchange between the components that comprise the application, an interconnection system has the following requirements:

1. to support the interconnection of existing tools and application components, requiring minimal modifications to each component.
2. to support heterogeneous computational models, corresponding to the multiple components of the applications.
3. to provide some degree of architecture and operating system independence, and easy retargetability.

1.2 Why current solutions are not satisfactory

Several alternatives to interconnect components are possible:

Using an existing message-passing system Most of the current message-passing systems do not fulfill the above mentioned requirements 1 and 2, as they do not have mechanisms to allow interprocess communication between separately started applications. For example, in MPI-1, the MPI intercommunicator mechanism allows the linking of processes in two different groups that may communicate using send and receive calls, but has limitations. For example, only pairs of processes can be interconnected and this must be managed explicitly in each component.

If we consider PVM, we see that requirement 1 is partially achieved. However, possibly some rewriting of the components would be necessary to prevent tag conflicts. Also, support of PVM everywhere would introduce unnecessary overhead.

Using a common interprocess communication mechanism The interconnection of components could be achieved by hand coding of the applications that interact, using some common interprocess mechanism (sockets, for example). This would require a knowledge of the particular interprocess communication and would not allow the reuse of code in other architectures.

1.3 How PHIS achieves the above requirements

PHIS achieves all the requirements above:

1: It has facilities for interconnecting separetely started components, and provides a well-defined and architecture-independent interconection programming model.

2: It allows the communication between components written using different programming models.

3: Its design eases the porting to a wide variety of architectures and runtime systems.

1.4 Organization of the paper

The paper is organized as follows. In section 2 an overview of the PHIS system is given, and its use is illustrated with a simple example. Section 3 describes the internal logical architecture of the PHIS system, the current implementation status, and its most distinctive aspects. In section 4, we show how PHIS is being used to implement a heterogeneous computational steering environment for genetic algorithm applications. A comparison between the functionalities of PHIS and other related systems is made in section 5. Finally we briefly present some conclusions and outline further work.

2 Overview of PHIS

An application is built of several *components*. Each component is an existing or new parallel program that is characterized by a *virtual machine* and an associated programming model (see figure 1). We assume, from the point of the view of the hardware, that the virtual machine can be supported on top of a workstation, a (possibly heterogeneous) network of workstations, a distributed–memory multiprocessor or a shared–memory multiprocessor, or any combination of the above. Regarding the programming model, we restrict (at least for the moment) our system to components that rely on a message–passing library, for example PVM [8] or MPI [6]. The application builder must have access to the source code

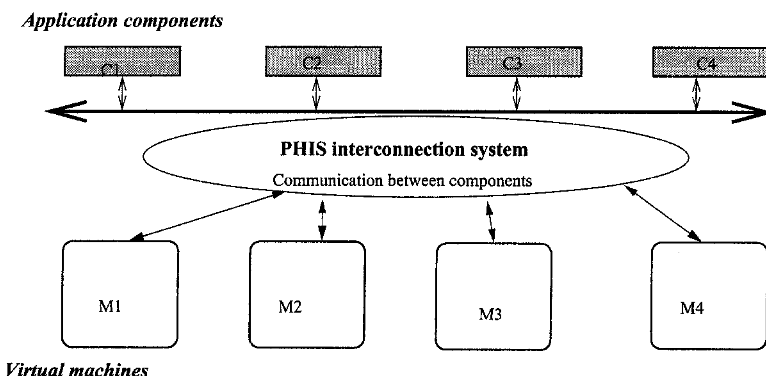

Fig. 1. System's overview

of the applications and must be able to introduce some modifications. We expect these modifications to be small and confined to well defined parts of the code.

2.1 PHIS programming model

We assume that an application is built from a statically-defined number of components. In designing the application, the programmer must identify, in each component, the processes that will interact with the other components. These processes will be the ones that must be modified and linked to the library that supports the communication and synchronization between components.

The interconnections between components are established before starting each component and cannot be modified during execution. The main concept for the integration of distinct components in the PHIS interconnection system is a specialized form of process group. The groups in PHIS are closed (only a process inside the group can send a message to the group [15]) and static (in the configuration phase processes enter in groups, and can only exchange messages after a configuration freeze). The process groups have a name that is an ASCII string. The sending of messages to groups is asynchronous [1]. By default, on message arrival a handler is called that enqueues the message for later processing. However, on entering the group, the process can specify an application-specific handling routine.

According to this, the modifications made to the source code of each component should concentrate on the following parts:

- On the initialization phase, the integration of the process into the relevant groups.
- Where appropriate, the sending and handling of interconnection messages.

A simplified description of the PHIS primitives follows:

Initialization	PHIS_Init(+ComponentName)
Joining a group	PHIS_Join(+GroupName, +HandlingRoutine)
Freezing the configuration	PHIS_Freeze( )
Termination	PHIS_Finalize()
Buffer setup for sending	PHIS_InitPackBuffer(−bufId)
Putting data in a buffer	PHIS_PackXXXX(+bufId, + data)
Sending the message	PHIS_Send(+GroupName, +bufId)
Message availability	PHIS_MessagesAvailable(+GroupName, −res)
Receiving a message	PHIS_Recv(+GroupName, −bufId)
Getting data from a buffer	PHIS_UnpackXXXX(+bufId, − data)
Buffer releasing	PHIS_FreeBuffer (+bufId)
Barrier	PHIS_Barrier(+GroupName)

2.2 Skeleton of the development of an application using PHIS

As an example of the PHIS functionalities we present a very simple application with two components, where one component calculates a Mandelbrot set and the other displays the result in a X-window display:

[1] In the following, the term message refers to the PHIS interconnection messages between application components, unless stated otherwise.

1. Component "calculate" runs on a PVM virtual machine, and was programmed using a computer farm approach (1 Master + N slaves). If we want to calculate the Mandelbrot set corresponding to a rectangle with heigth H, and length L, a natural approach to the problem is to divide this rectangle in several strips and make each strip a unit of work. Each worker receives a strip defined by its width, height and the coordinates of lower left and upper right vertexes. A calculation of a bit map, according to a recurrence formula is performed. The bitmap obtained is the result of the workers' work.
2. Component "display" is a sequential process that runs on a workstation.

Figure 2 shows the interconnection between the two components.

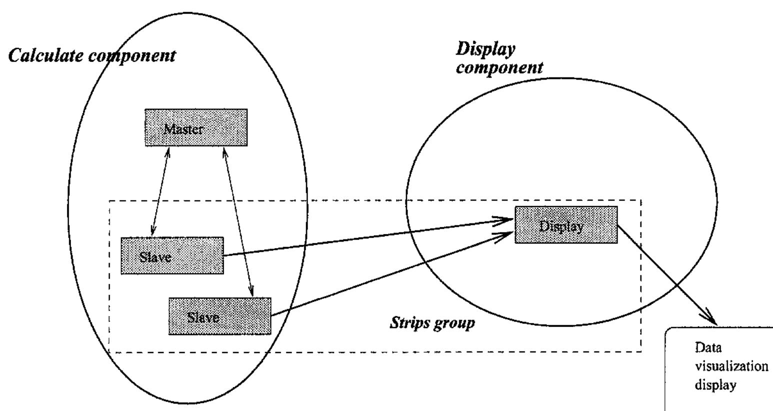

Fig. 2. Components in the Mandelbrot application

In the following we present the three programs that compose our system, including the calls to the PHIS system.

```
/* ——— master program ————*/
main() {
   pvm_spawn('slave', ..., NSLAVES, tids)
   divide the space in M tiles
   for (i=-1; i < M; i++){
     /* wait for a request and reply with TILE i */
     pvm_recv( -1, ... ); pvm_bufinfo( ..., &tid, ... )
     pvm_initsend(...) ; pvm_pkfloat(...); pvm_send( tid, ... )
   }
   for (i=0; i < NSLAVES; i++){
     /* all tiles sent: wait for the slave to request a tile and kill it */
        pvm_recv( -1, ... ); pvm_bufinfo( ..., &tid, ... ); pvm_kill(tid)
   }
   pvm_exit()
}
```

```
/*------- slave program -------*/
main() {
  mytid = pvm_mytid(); myparent = pvm_parent()
  PHIS_Init('calculate'); PHIS_Join('strips', DefaultHandler);
  PHIS_Freeze();
  do{
     /* ask for a tile */
     ... pvm_send( myparent, ...)
     pvm_recv( -1, ... ); pvm_upkfloat( ... ); ... ;
     calculate_tile(...);

     PHIS_InitPackBuffer( &b ); PHIS_PackByte(b,...); ... ;
     PHIS_Send('strips',b);
  }while (1);
}

/*---------- display program -----------*/
main() {
  PHIS_Init('display'); PHIS_Join('strips', DefaultHandler);
  PHIS_Freeze();
  do{
     PHIS_Recv( 'strip', &b );
     PHIS_UnpackByte(b,...); ... ;
     display the part of the pixel map received
     PHIS_FreeBuffer(b);
  }while (1);
}
```

3 PHIS architecture

The PHIS system is internally organized as a distributed architecture consisting of a collection of cooperating daemons that support the communication primitives described in 2.1. Each daemon process supervises the interaction between a single application component and the other components in a PHIS configuration. Such interconnection daemons (called *PHIS agents*) run in one of the processors belonging to the virtual machine that supports the execution of its associated application component. The basic functions provided by each PHIS agent are supported through the following two libraries:

Intercomponent library This library supports the communication among the daemons. As we suppose that the interconnection between the components is supported by a "software bus" made of a local network running the TCP/IP protocols, there is no difficulty in using the well known facilities of these protocols (sockets, RPCs, multicast) to build this library.

Intracomponent library This supports the commmunication between each PHIS agent and the processes which constitute its associated application component. The messages sent by these processes must be forwarded to the component's outside environment, and any incoming messages must be

delivered to the destination processes in the component (i.e. those belonging to the message destination group).

Figure 3 illustrates the relationship between each PHIS agent and the processes in an application component. The implementation of these libraries is not

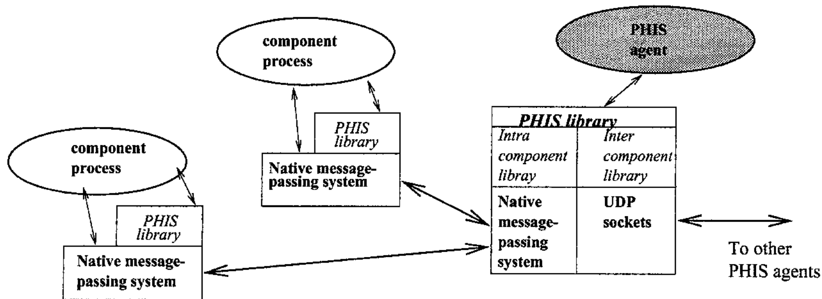

Fig. 3. Runtime environment of processes in each component

further discussed here due to lack of space. However, we must stress the fact that they are implemented with portability in mind. For example, the intracomponent layer is internally decomposed into two layers, one that is architecture-independent and the other that depends on the native communication system. A well–defined interface between the two layers exists.

3.1 Configuration of an application

An application is configured through a *configuration file* that is visible to the application processes that interact with other components. The configuration includes, for each component, the following information:

- Name of the component (as given in PHIS_Init()).
- Number of processes in the component that must interact with other components.
- Name and architecture of the machine where the interconnection daemon should be launched. This physical machine belongs to the virtual machine where the component runs.
- A character string describing the communication protocol that should be used to communicate between the interconnection daemon and the components of the application (for example "sysVipc", "tcp", "parix").
- A character string with information relevant to the above mentioned protocol in the previous field (for example a TCP/IP port, a shared memory key for UNIX System V IPC, etc.).

The configuration file for the Mandelbrot application is given below:

```
;
; application processes arch  name     protocol parameters
;
calculator    8         sunos frodo    tcp      2500
display       1         aix   grafikus sysVipc  333
```

3.2 Integration of the PHIS calls in already existent applications

When we modify the source code of a program that is executed as a process belonging to a component we introduce calls to the programming interface described in 2.1. A question that immediately arises is the possibility of interference between the calls made by application processes to the original message–passing system and the ones internally used by the interconnection system. As the interconnection system uses asynchronous sending primitives, the main question concerns the receiving of messages. As most of the hardware platforms support the UNIX system call interface, the most natural way of handling the arrival of messages from the interconnection system is to associate this event with the delivery of a signal. The handler will receive the message, and the only requirement that must be made to the original message–passing system is to be "signal-safe".

3.3 Implementation status

The ongoing implementation of PHIS runs on a set of UNIX workstations. The mappings of the PHIS architecture to this physical architecture are as follows:

- the network of agents is initialized using the 'rsh' facilities;
- the agents communicate using a reliable protocol built on top of UDP;
- the primitives of the intercomponent library are implemented using UNIX system facilities and sockets;
- we are experimenting with components writen using PVM (which is signal–safe) and the MPICH implementation of MPI.

4 Using PHIS to support a heterogeneous computational steering environment

In a related ongoing project we are currently developing an environment supporting parallel and distributed computational steering of applications based on genetic algorithms. The project aims at producing a highly efficient and productive environment that can be used by researchers in a department of environmental sciences, as well as in our department, in order to solve a large diversity of optimization problems. One requirement posed by this environment concerns its efficiency which can only be fully achieved through parallelism. Three prototypes for parallel genetic algorithms have already been developed towards this goal:

- A prototype supports the parallel execution on a shared-memory Pentium-based multiprocessor, under the WindowsNT operating system, using a master-slave model as a basis [12].
- A prototype supports parallel and distributed execution on a PVM platform, using the island model as a basis [4].
- A prototype integrates genetic algorithms and simulated annealing and supports parallel execution on the PVM system [5].

The environment consists of the following components:

1. A computationally intensive component supports the parallel execution of the genetic algorithm; its interface with other components in the environment is defined by a set of parameters which characterize the genetic algorithm, and the description of a population of individuals.
2. A visualization component supports on-line data visualization of the outcome of the computational component in a graphic workstation; it also supports visualization of performance data, obtained by having a monitoring component associated with the computational component.
3. An interactive control component allows the dynamic modification and inspection of the application parameters whose values can also be displayed.

The above mentioned prototypes and components illustrate another important requirement posed by this environment, namely concerning the need to support heterogeneity, not only at the parallel platform level, but also regarding the parallel computational models. We are using PHIS as an intermediate-level infrastructure that supports the above requirements, as illustrated in figure 4, where we show a PHIS configuration with 4 components:

- The GeneticAlgorithm component, which encapsulates a parallel genetic algorithm that internally uses the master-slave model and is implemented on top of PVM.
- The DataVisualization component is responsible for data interpretation and display.
- The PerformanceVisualization component which is responsible for application monitoring and display of performance-related data. The performance visualization components rely upon existing tools such as Paragraph [10] coupled with a distributed monitoring system [2].
- The Control component, which is responsible for the application of the steering commands, and is actually being based on a distributed debugger (called DDBG) for PVM that we have independently developed on a related project [1] [2].

The definition of this PHIS configuration requires the specification of the following PHIS groups:

Data Visualization Group The Master process and the process(es) responsible for the data visualization are members of this group.

[2] Alternatively in figure 4 we could have two separate Monitoring and Performance-Visualization components, easily configured using PHIS.

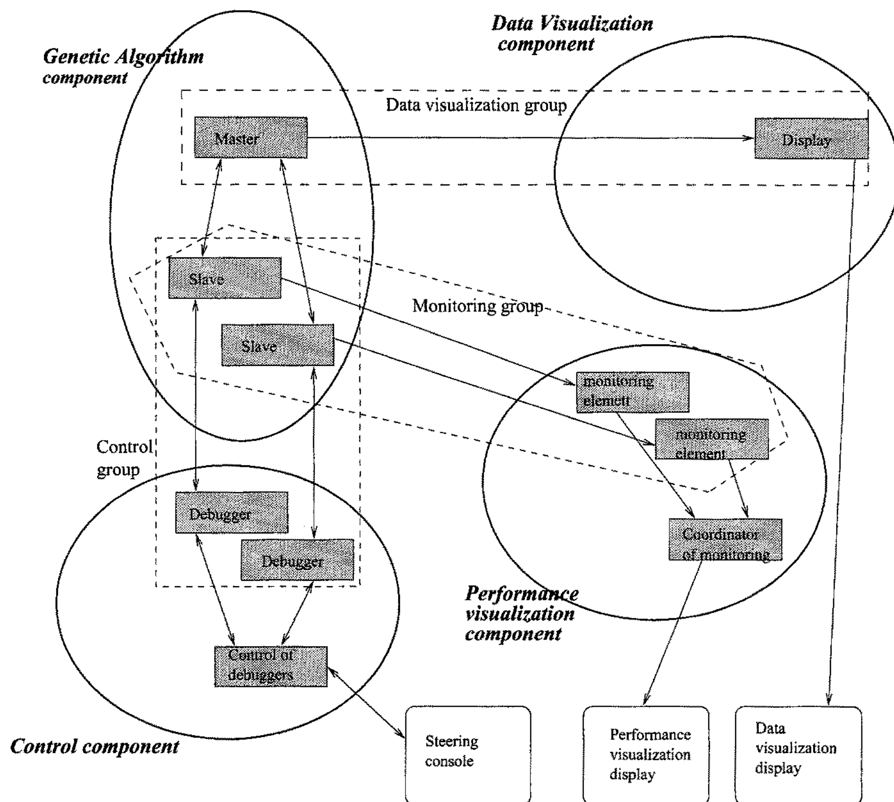

Fig. 4. PHIS in a computational steering environment for genetic algorithms

Monitoring Group The monitoring group consists of N distributed monitoring processes, each associated with a distinct slave process, and a coordinator. This group is responsible for the local recording, transfering and gathering of performance information.

Control Group The slave processes and a debugger system allow the modification of the genetic algorithm parameters during the execution.

5 Related work

The construction of applications through the integration of multiple software modules executing on a heterogeneous set of machines has been recognized as an useful paradigm for some time [13]. One pioneering effort has been the Schooner system [11]. Recently, this area has received contributions by several projects like CUMULVS [14], Legion [9], CAVEComm[3] and I-Way/Globus [7].

The major advantage of PHIS (and other related systems like the ones mentioned above) is the support of communication between applications written us-

ing different parallel programming models, e.g. a PVM-based component and a MPI-based component. The programming model of PHIS is more flexible than many of the related models cited above. For example Schooner forces a model of component interconnection based on RPCs with the known advantages and disadvantages. The main disadvantage is, in our opinion, the impossibility of multicasting. Some of the other systems are too connected to a particular message-passing system (like CUMULVS with PVM) or don't have a high level interconnection model (for example I-Way suggests the use of TCP/IP sockets for the interconnection of components). The PHIS programming model has some similarities with CAVEComm.

Another distinctive feature of PHIS is its concern with the portability. To port PHIS to a different runtime system it is only necessary to rewrite the machine-dependent part of the intracomponent library.

6 Conclusions and further work

We have described the current status of an ongoing project that is developing the PHIS system, an infrastructure to support the interconnection of distinct application components. Due to the large diversity of parallel computational application components and support tools that have been developed in the last few years, using distinct programming models, and parallel platforms, we think there is a very strong motivation to this work. This is confirmed by our own experience in our faculty campus, where PHIS is showing great flexibility to support the interconnection of already developed components, as described in section 4. Currently all these components rely on the PVM system, but we do not antecipate major difficulties concerning the coupling of our interconnection libraries with MPI applications. Future work will include:

- to port PHIS to other architectures in order to exploit the heterogeneous platform existing at our Department, which includes UNIX workstations, a DEC Alpha-based FDDI cluster, and two Transputer-based multicomputers.
- to evaluate the overhead of using PHIS when compared to solutions where all the components use the same runtime system.
- to use PHIS to interconnect components that are not message-passing based, namely ones that use a data-parallel approach and ones that use the shared virtual memory paradigm.

Acknowledgments

This work was partly suported by the CIENCIA and PRAXIS XXI (project PROLOPPE) Portuguese Research programmes, the EEC Copernicus and TEMPUS programmes and DEC EERP PADIPRO project.

References

1. Cunha, J., Krawczyk, H., Wiszniewski, B., Mork, P., Kacsuk, P., Luque, E., Sutovska, L., and Hluchy, L. Monitoring and debugging distributed memory systems. In *Proceedings of uP94, Eight Symposium on Computer and Microprocessor Applications* (Budapest, 1994).
2. Cunha, J. C., Lourenço, J., and Antão, T. A Debugging Engine for a Parallel and Distributed Environment. In *Proceedings of DAPSYS'96, 1st Austrian-Hungarian Workshop on Distributed and Parallel Systems* (Misckolc, Hungary, Oct. 1996).
3. Disz, T., Papka, M., Pellegrino, M., and Szymanski, M. CAVEComm users manual. Tech. Rep. ANL/MCS-TM-218, Math and Computer Science Division, Argonne National Laboratory, September 1996.
4. Duarte, L., and Duarte, J. Genetic algorithms and parallel processing. in portuguese, projecto final de licenciatura, 1996.
5. Fert, G. Genetic annealing and parallel genetic annealing. Master's thesis, University of Wroclaw / Universidade Nova de Lisboa, 1996.
6. Forum, M. P. I. MPI: A message-passing interface standard. Tech. Rep. Computer Science Department Technical Report CS-94-230, University of Tennessee, Knoxville, TN, May 5 1994.
7. Foster, I., Geisler, J., Nickless, B., Smith, W., and Tuecke, S. Software infrastructure for the I-WAY high performance distributed computing experiment. In *Proceedings of High Performance Distributed Computing 1996* (1996).
8. Geist, G., and Sunderam, V. Network-based concurrent computing on the PVM system. *Concurrency: Practice & Experience 4*, 4 (June 1992), 293–311.
9. Harper, R. Interoperability of parallel systems: Running PVM in the Legion environment. Tech. Rep. CS-95-23, Dept. of Computer Science, University of Virginia, May 1995.
10. Heath, M. T., and Etheridge, J. A. ParaGraph: a tool for visualizing performance of parallel programs. University of Illinois and Oak Ridge National Laboratory, January 1992.
11. Homer, P. T. *Constructing scientific applications from heterogeneous resources.* PhD thesis, University of Arizona, December 1994. Department of Computer Science.
12. Horta, B. Optimization using genetic algorithms and parallel processing. in portuguese, projecto final de licenciatura, 1994.
13. Khokhar, A. A., Prasanna, V. K., Shaaban, M. E., and Wang, C. Heterogeneous computing challenges and opportunities. *IEEE Computer J. 26* (June 1993), 18–27.
14. Kohl, J., and Papadopoulos, P. CUMULVS user's guide: Computational steering and interactive visualization in distributed applications. Tech. Rep. ORNL/TM-13299, Computer Science and Mathematics Division, Oak Ridge National Laboratory, August 1996.
15. Liang, L., Chanson, S., and Neufeld, G. Process groups and group communications: Classifications and requirements. *IEEE Computer* (Feb. 1990), 56–65.