ELSEVIER

Future Generation Computer Systems 18 (2002) 317–333

FGCS
FUTURE GENERATION COMPUTER SYSTEMS

www.elsevier.com/locate/future

MPI–Delphi: an MPI implementation for visual programming environments and heterogeneous computing

M. Acacio*, O. Cánovas, J.M. García, P.E. López-de-Teruel

Dpto. de Ingeniería y Tecnología de Computadores, University of Murcia, Campus de Espinardo, s/n 30080 Murcia, Spain

Abstract

The goal of a parallel program is to reduce the execution time, compared to the fastest sequential program solving the same problem. Parallel programming is growing due to the widespread use of network of workstations (NOWs) or powerful PCs in high-performance computing. Because the hardware components are all commodity devices, NOWs are much more cost-effective than custom machines with similar technology. In this environment, the typical programming model used has been message-passing and the MPI library has become the standard in the distributed-memory computing model. On the other hand, visual programming environments try to simply the task of developing applications. They provide programmers with several standard components for creating programs. Delphi constitutes one of the most popular visual programming environments nowadays in the Windows market place. In this paper, we present MPI–Delphi, an implementation of MPI for writing parallel applications using Delphi visual programming environment. We show how MPI–Delphi has been developed, and how it makes possible to manage a cluster of homogeneous/heterogeneous PCs. Two examples of use of MPI–Delphi in a heterogeneous cluster of workstations with a mixture of Windows and Linux operating systems are also included. The MPI–Delphi interface is suitable for some specific kinds of problems, such as monitoring parallel programs of long execution time, or computationally intensive graphical simulations. In addition, MPI–Delphi has proven to be a good tool for research, as the development of new algorithms can be carried out quickly and, therefore, time spent on the debugging of such algorithms is reduced. Finally, we conclude by explaining some of the tasks we think MPI–Delphi is suitable for.

Keywords: Network of workstations/PCs; Parallel visual programming environment; MPI for Delphi; Graphical user interface; Heterogeneous computing

1. Introduction

In recent years, parallel systems and parallel programming have increasingly made use of message-passing paradigm, in order to solve complex computational problems. Although the notion of network of workstations (NOWs) is not new, NOWs or PoPCs are currently being considered as a cost-effective alternative to the use of expensive, dedicated high-performance systems. In fact, the increasing performance and availability of general-purpose microprocessors and networks has fostered the spread of NOWs, either with shared or distributed-memory, as an alternative to customized massively parallel

* Corresponding author.
E-mail addresses: meacacio@ditec.um.es (M. Acacio), ocanovas@ditec.um.es (O. Cánovas), jmgarcia@ditec.um.es (J.M. García), pedroe@ditec.um.es (P.E. López-de-Teruel).

0167-739X/02/$ – see front matter
PII: S0167-739X(01)00054-1

systems (MPPs). So, a cluster of powerful PCs interconnected with a fast network could be seen as a reasonable and cost-effective alternative approach [7,23] for exploiting parallelism.

Due to the rapid advance in performance of these commodity computers, when such clusters are upgraded by addition of other nodes, they become heterogeneous. We use heterogeneous in two senses: (a) computers with different classes of properties, such as microprocessors, memory RAM size, and so on, and (b) computers with different OS, mainly UNIX (with its variants) and Windows NT/2000. Currently, many organizations have large and heterogeneous collections of networked computers. Therefore, the question of managing these computers efficiently is now a major problem.

Traditionally, Linux/UNIX has been used as the operating system in clusters of PCs. However, Windows 95/98/NT is currently perhaps the most widely used operating system. Moreover, the Intel Pentium II/III, together with its Windows 98/NT/2000 operating system (the Wintel model), provides enough overall capacity to also displace the RISC/UNIX workstation in the engineering and scientific marketplace. As a result, there are more and more clusters of PCs using Windows 95/98/NT/2000 as their operating system [5,21,26]. The advantage of this operating system over Linux/UNIX is greater ease of use, and a larger number of developed applications. Among these applications, programming environments are not an exception.

Lately, some software distributed shared-memory (SDSM) systems have appeared allowing shared-memory parallel programming in NOWs with Linux/UNIX [3,11] and Windows [12,27] operating system. However, the main problem in this configuration is the high cost of access to shared-memory. This cost precludes these systems from being widely used in this environment. Therefore, message-passing paradigm continues to be the preferred paradigm in NOWs.

With the advent of the MPI [10] standard, parallel programming using the message-passing style has attained a certain level of maturity. However, in terms of convenience and productivity, this programming model suffers from low performance of the generated code, due to the lack of high-level development tools. Although, in recent years a wide range of tools of this type have been developed, the situation is still not satisfactory for users, since most of these tools can only be used in isolation and cannot work in heterogeneous environments.

The major obstacle to program applications for parallel systems is the programmer's ability to deal simultaneously with the complexity and parallelization of the algorithms under consideration, and the details of communications between processors. We believe that it is necessary to devise appropriate visual programming environments to overcome this difficulty.

Nowadays, most programming projects are developed using visual environments. There is a great variety of programming environments available for Windows. Visual programming environments constitute one of the most interesting development tools. They make the creation of programs easier for the programmer, as they provide programmers with the components necessary for developing applications. The work of the programmer is to adapt these components to the specific applications. In this way, the creation of certain parts of the application (such as the user interface) is greatly simplified.

Delphi[2] constitutes one important visual programming environment. Unlike other visual programming environments, Delphi is a real compiler, which generates very efficient executable code, with no need to distribute runtime additional files with the application. So, it is of great interest to implement parallel applications with Delphi, in order to take advantage of visual programming facilities. This would not only accelerate the implementation of parallel programs, providing programmers with an easy way of showing graphical information, but may also be used for debugging new computation-intensive algorithms.

The first and most obvious problem that we encountered was the lack of an MPI implementation for Delphi. This visual programming environment uses Object Pascal (an object-oriented version of Pascal) as its native programming language, while available Windows MPI versions were accessible only from Visual C++ [1] and Borland C++. [2]

In this paper, we describe an implementation of MPI for Delphi named MPI–Delphi. Our central objective was to give MPI functionality within this

[1] Visual C++ is a trademark of Microsoft.

[2] Delphi and Borland C++ are trademarks of Inprise (formerly Borland).

popular environment, using one of the existing Windows MPI implementations. So, MPI–Delphi allows us to develop parallel applications using Delphi and making good use of the facilities that it provides for applications with many user interactions and/or complicated graphical requirements. A preliminary publication of this paper can be found in [2].

MPI–Delphi has two main features. Firstly, it gives us an MPI implementation for Delphi; this is, to date, the only implementation for Pascal-style language we know. MPI–Delphi is a visual programming environment that allows the user to manage a large cluster of homogeneous/heterogeneous PCs. In this way, MPI–Delphi — due to the fact that it is based in the MPI standard — is a powerful tool for developing and executing parallel applications, which can have diverse computational requirements.

We have structured this paper as follows: first, we describe how MPI–Delphi was created (the problems we encountered in its development, and how these were solved), and then we describe the parallel environment that we propose. For this purpose, we show a sample application created using MPI–Delphi, along with the visual results obtained for it. The application consists of the implementation of the well-known Jacobi relaxation method, applied to solve the problem of the heat diffusion in a body. We called this program Visual Jacobi, as it incorporates a good user interface, in which we can easily modify the values of the input parameters, as well as observe the evolution of the state in the body. The main goal of this sample is to visualize the evolution of the temperature. This graphical information was easily shown, thanks to Delphi's graphical components. In the next section, we study the use of MPI–Delphi to develop a scientific application: the parallel visual version of the EDR algorithm, a new method for estimation of probability density functions from uncertain samples. In this case, we employed MPI–Delphi to assist us in debugging the heuristics of this new algorithm. Thanks to MPI–Delphi, we were able to use the visual components to validate the heuristics, and to reduce the debugging time. Moreover, for this algorithm we show an MPI–Delphi performance analysis. The results are very promising, both in terms of debugging time of the algorithm and in terms of the overhead introduced by the MPI–Delphi environment itself. We then mention some related work. The final part of this paper is an outline of the conclusions of our work, and several future possibilities.

2. The development of MPI–Delphi

MPI–Delphi is the name of the implementation of MPI for Delphi we have developed. We have tried to keep our implementation as close to the C specification as possible, but some minimal modifications have been introduced.

Currently, several MPI implementations for Windows can be found. There are implementations of these libraries for Windows 3.1, 95/98 and NT/2000 systems. The common characteristic we have found in all these versions is that the language they have been created for is C (or C++), and the environments from which they can be used are Visual C++ or Borland C++. Moreover, there is a proposal, named *mpiJava* [4], which provides MPI capabilities to Java developers (*mpiJava* offers a suitable class hierarchy based on MPI-2 standard). However, we are not aware of any implementation of MPI available for Delphi developers, despite the great capabilities that this IDE (Integrated Development Environment) offers.

We have used WMPI v.0.9b [3] as the basis to develop a mechanism to provide MPI functionality to Delphi. We made this selection taking into account several factors such as reliability, cost, and rapid development of new improved versions. WMPI is a full MPI standard implementation for Microsoft Win32 platforms. It is completely compatible with MPICH 1.0.13 [4] (which is also a free MPI implementation from the Argonne National Laboratory) and it uses P4 message-passing as its underlying protocol. Connectivity between WMPI and other clusters running MPICH 1.0.13 with distinct operating systems (several UNIX and Linux versions, as well as Windows 32-bit platforms) is accomplished through a common TCP/IP network. So, by using the MPI standard, clusters of heterogeneous computers can be managed as a single image in an easy way.

As we commented above, WMPI is available for Visual C++ and Borland C++ environments.

[3] WMPI: http://dsg.dei.uc.pt/~fafe/w32mpi/intro.html (currently, the 1.5 beta version is available).

[4] MPICH/NT: http://WWW.ERC.MsState.Edu/labs/hpcl/projects/mpi/mpiNT-download.html.

Although, Delphi and Borland C++ are products from the same company, programming libraries for Delphi are not compliant for C++, and vice versa. For this reason, it is necessary to provide a mechanism to offer Delphi programmers a way to use WMPI libraries. This section describes the way we carried out the implementation of this mechanism.

2.1. First step: providing Delphi with MPI capabilities

Initially, the first problem was to find a way to communicate programs developed with Delphi and the WMPI library. In order to solve this first problem, we evaluated several mechanisms, paying attention to certain criteria, such as feasibility, efficiency, commodity, transparency and scalability. Thus, in the implementation of MPI–Delphi, we set ourselves two main objectives:

1. To adapt C or FORTRAN definitions of MPI functions to Pascal syntax, given that MPI has been defined for C and FORTRAN languages.
2. To carry out an implementation in which the functionality of W32MPI was offered to Delphi in an easy way. Delphi programmers should be able to make use of the MPI functionality in a simple, transparent way, without having to worry about the internal details of the implementation. Users should only need to know how to use the MPI standard functions.

The first objective is related to the MPI specification. C programming language could be considered nearer to Pascal than FORTRAN, so we used the C specifications of MPI as our basis. However, independently of the mechanism used to connect Delphi with WMPI, there is an obvious problem when trying to translate C to Pascal: the differences between C and Pascal. They are two very different languages in some aspects such as data types. This problem is outlined in the next section.

In order to deal with the second objective, two solutions were analyzed:

1. The use of DDE (dynamic data exchange) to communicate W32MPI and Delphi.
2. Creating a DLL (dynamic link library) providing MPI functionality.

DDE is a message-based protocol that allows information interchange between Windows applications. DDE uses a client/server model in which the application requesting data is considered the client and the application providing data is considered the server.

DDE applications use a three-tiered identification system to distinguish each one from other DDE applications.

- *Application Names* are at the top of the hierarchy and refer to a server application.
- *Topic Names* further define the server application. A server can support more than one topic.
- *Item Names* identify details within a topic name, and each topic can have more than one item.

The DDE Application Name is almost always the executable filename for the server application (without the EXE extension). The DDE Topic typically identifies a group or category of data in the server application, and each data item that a server can provide has a unique DDE Item Name. Thus, Application Name, Topic, and Item Name identify the exact source of the data in a server application.

The data communication is carried out using blocks of global memory. Servers send to their clients a pointer to those memory blocks, not the information itself. There are three different ways data exchange can occur.

1. A client application can request data all at once from a server application.
2. A server application can send data to a client application.
3. A server application can advise the client application that an item has changed its value.

A DDE-based solution would involve a server application written in C++, offering MPI functionality to client applications written in Delphi. Delphi clients would establish connections with this server application, which should run before Delphi parallel programs, in order to gain access to WMPI functionality. This scheme corresponds to Fig. 1.

This proposal has several drawbacks.

- First, the client must be able to send information to the server, and with DDE this is not possible (clients can only specify notification modes and application/topic/item). In order to carry out such

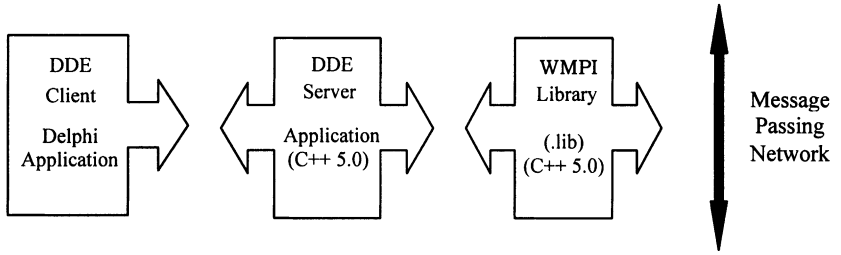

Fig. 1. Communication between the elements of DDE-based solution.

communication, client applications must be both clients and servers.
- Second, if we considered the large range of data types and functions supported by MPI, a structure based on topics and items would be extremely complex and impractical.
- Third, as an additional application is necessary (DDE Server), parallel Delphi programs are overloaded with DDE functionality, and MPI–Delphi scalability is compromised.

Therefore, we turned our attention to the other alternative. DLLs are Windows-based program modules that can be loaded and linked at runtime. They can contain commonly used routines and resources that Windows and Windows applications call up when they need them. Some benefits of DLLs are the following:

1. Minimize the application's resource requirements.
2. Make code distribution more flexible.
3. Give the code a form of language independence, which means that a DLL created in C++ (or another language) can be accessed by most other programming languages, such as Delphi.

Our final choice was determined by feasibility, efficiency, commodity, transparency and scalability. Thus, we concluded that the option based on DLLs represented the most suitable solution. This option also involves using C++ in order to create the DLL.

Thus, we have implemented a DLL using Borland C++ 5.0 which integrates almost all the functionality present at WMPI. This DLL can be accessed from any Delphi application, paying attention to the conversion between Delphi and C variables.

This approach constitutes an elegant solution, and it represents a simple and structured way to carry out the communication between Delphi applications and MPI. Delphi programmers only have to learn how to use the DCU (Delphi compiled unit) which provides access to our DLL (in a transparent way for such developers). The programmer simply uses the functions contained in the DCU in the same way as when using any other DCU.

2.2. Second step: development of the DLL

As we have just seen, a DLL is the best option to accomplish our task. Thus, we have developed a DLL in Borland C++ 5.0 named *dllmpi.dll*. This section describes the most important issues related to the implementation of *dllmpi.dll*, such as specification of exported functions, and compatibility of data types. This last issue is a complex task, given the important differences between the flexible data-typing system of C language and Delphi's strict system.

Firstly, it must be noted that this DLL does not offer a full MPI implementation, i.e. it does not export the whole functionality provided by the MPI standard: communicators have been omitted in this first version of MPI–Delphi. The reason is that the main goal of this version is to determine the possibilities of parallel programming using Delphi, more than to implement the whole functionality of the standard. Therefore, and for the sake of simplicity, the communicator argument was eliminated in the implemented DLL. In any case, MPI–Delphi supports completely all the operations related to point-to-point communications and collective communications, initialization and termination, control, error management, and so on.

All of the exported functions in our DLL have three common elements (words in bold):

```
int FAR _export MPI_Finalize_beta(void)
```

- **FAR**. It is a scope modifier which specifies that functions containing it could be invoked from another memory segment.
- **_export**. It is used to export the function.
- **_beta**. It is a suffix added to avoid name conflicts (there is a function called MPI_Finalize in WMPI, and *dllmpi.dll* includes WMPI library). In our DLL all the functions have this suffix.

The body of some functions in *dllmpi.dll* is very simple, consisting only of an invocation to the WMPI original function. However, in most cases it is not so simple. The serious drawback of DLLs is that they lack type checking. When a DLL is created, its functionality is exposed. Then, the developer's compiler using a function from our DLL is faced with two problems: how to verify the correct number of parameters; and how to check the parameter types.

To design this DLL, we began with a complete study of the MPI standard. It was necessary to understand the kind of functions, data types and other aspects related to MPI standard. Using the MPI standard definition [10] and WMPI headers (.H files), we determined the elements that our DLL must offer.

However, in order to implement some functions, we have used a technique based on the definition of intermediate data types. There are some functions which work with complicated data types, or data types not clarified after our preliminary study of the MPI standard. In these cases, we define intermediate data types inside the DLL, to carry out assignments between these new data types and such unknown data types. Assignments are performed using the well-known fields, not copying the whole structure of data types as occurs in functions where there are no problems with the data types involved. Some examples of these exceptional data types are *MPI_Status*, *MPI_Op*, and *MPI_Datatype*. In order to illustrate this concept, Fig. 2 includes the definition of a well-known data type, and an exchange of information with an unknown type.

2.3. Third step: development of the DCU

Once the DLL is implemented, we must decide how Delphi programmers can to carry out MPI applications. One possibility is to use our DLL directly, i.e., including it in their projects. As we commented above, the Delphi compiler has no way of checking whether the exported functions are named with the correct number and type of parameters. The way to solve this problem is by providing a mechanism to hide the existence of DLL. There are several

```
//We define a data type as close to the original MPI_Status data type as possible

struct Status {
       int Count;          /* Received bytes counter */
       int MPI_SOURCE;     /* Message source */
       int MPI_TAG;        /* Handler id */
       int MPI_ERROR;      /* Error */
};

typedef struct Status DelphiStatus;

//Implementation of MPI_Wait_beta function

int FAR _export MPI_Wait_beta(MPI_Request *request, DelphiStatus* stat)
{
       MPI_Status status; //MPI_Wait needs a MPI_Status variable
       int Result; //Result from MPI_Wait

       Result = MPI_Wait(request,&status); //We call the original MPI_Wait
       StatusToDelphi(&status,stat);//Copy of values field by field
       return Result;
}
```

Fig. 2. Data exchange between original and intermediate data types.

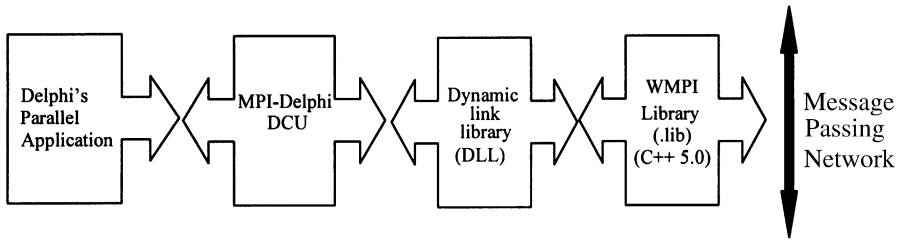

Fig. 3. Communication between the elements of MPI–Delphi.

alternatives, which Delphi supports, and which could accomplish this task, ranging from components to DCUs. We decided to develop a DCU including data types and functions related to MPI standard. The DCU offers Delphi programmers a similar way of writing applications to that found in the MPI API for C language. Delphi programmers include in their projects a unit (library or DCU) which provides MPI functionality.

Another important decision in the DCU design is not to modify the way a programmer uses MPI. Our DCU could have offered a set of classes in order to give an object-oriented approach to MPI [9,18]. However, we preferred to develop a unit as close to MPI standard as possible, trying to keep Delphi and UNIX programming of parallel applications as similar as possible. As we shall see in the next section, we propose a scheme based on a mixture of Linux/UNIX and Delphi processes collaborating on the same parallel problem, i.e. a heterogeneous system.

Fig. 3 shows the elements involved in the creation of MPI–Delphi. For example, if a program written in Delphi invokes *MPI_Finalize* function (contained in the DCU), this, in turn, invokes *MPI_Finalize_beta* included into the *dllmpi.dll* file. Finally, *MPI_Finalize_beta* calls up *MPI_Finalize* from WMPI. Thus, Delphi programmers have access to WMPI functionality with no direct knowledge of the intermediate DLL.

However, *MPI_Finalize* is a function with no parameters, and it is very easy to make it available to Delphi. There are other functions, which have more complex parameters of MPI data types. In such cases, our DCU defines data types with a structure based on the original MPI standard definition for C language of corresponding data types. This is possible because Delphi incorporates primitive data types identical to primitive C data types, such as *single* (*float*), *double*, *integer* (*int*), *longint* (*long int*) and so on. In this way, we have a correspondence between C and Pascal data types, and thus it is possible to interchange information between Delphi and WMPI, using the DLL.

There is a further question related to MPI programming from Delphi. Delphi developers are accustomed to following a particular style of programming: Delphi style. For this reason, our DCU introduces three small modifications:

1. *MPI_Init* (*int argc*, *char* *argv*): *argc* and *argv* arguments are passed directly inside the DCU, so we free the Delphi programmer from this task.
2. Some modifications regarding data types. Many MPI–C functions return a value representing a Boolean as an integer value (i.e. *MPI_Test*). In Object Pascal, a Boolean data type is available, so it is used as the returned type in this kind of functions.
3. Terminology applied to pointers. Our DCU replaces (*void**) parameters by *Pointer* data types.

Finally, it is important to mention that parallel programming of MPI-based applications using MPI–Delphi is very close to C specification for MPI. Given the fact that differences are minor, developers do not have to spend much time adapting existing MPI applications written in C.

3. The MPI–Delphi visual environment

In this section, we introduce the MPI–Delphi visual environment. This environment allows the user to manage a cluster of homogeneous or heterogeneous computers. First, we show how MPI–Delphi must be configured, and then we use a well-known scientific

application to describe the exact management of our environment.

3.1. The MPI–Delphi structure

MPI–Delphi can be used in a cluster of workstations using Windows 95/98/NT/2000 in each PC of the cluster (homogeneous system). However, W32MPI v.0.9b, and therefore MPI–Delphi, is compatible with MPICH v.1.0.13 for Linux/UNIX and Windows, so it is possible to combine processes in execution in all these operating systems. We propose a solution to take advantage of both aspects: performance, in terms of speed, of Linux/UNIX/NT (both in communication and processing tasks), and programming facilities of Windows.

The MPI–Delphi structure is based on two kinds of processes:

1. A single Windows process constituting the user interface, and which would contain all the graphical part of the parallel application. This process should be written using Delphi and MPI–Delphi. Its function is the distribution of the data involved in the problem to the rest of the processes, and to show, in a visual way, the evolution of computations carried out by the processes. In addition, it can be used to illustrate some statistical data, such as execution time, start-up time, etc.
2. The rest of the processes, written in C language, and used to perform the computations. First, these processes receive the problem configuration, and periodically must send the intermediate results of their computations to the Windows process. These processes are executed under Linux/UNIX/Windows NT/2000 operating systems, depending on speed performance criteria.

Note that this way of developing applications involves using two different programming languages: C and Object Pascal. However, the proposals for the two types of processes are very different: the first one (Delphi programmed) has to manage the graphical interface and the user interaction, while the rest of the processes (C programmed) manage the heaviest computational load (and therefore suitable for parallelism) of the specific algorithm implemented. So, it is perfectly appropriate to develop them in different languages, each one suitable for each kind of task. Fig. 4 shows the MPI–Delphi structure.

3.2. An example of use: Visual Jacobi

Visual Jacobi implements the classic parallel Jacobi relaxation algorithm [15] (body division by rows, with these groups of rows being calculated by different processes), but in addition, it also shows graphically the evolution of the body. Fig. 5 shows the two parts of the application code, both using MPI: on the left, written in C language, the main code to implement the parallel Jacobi relaxation algorithm, in a Linux/UNIX/NT environment; on the right, written in Object Pascal language, the code of the process that starts the parallel execution, and shows the results graphically.

The main advantage of the Delphi process is that interaction is user-friendly, allowing us to observe the evolution of the partial results obtained through

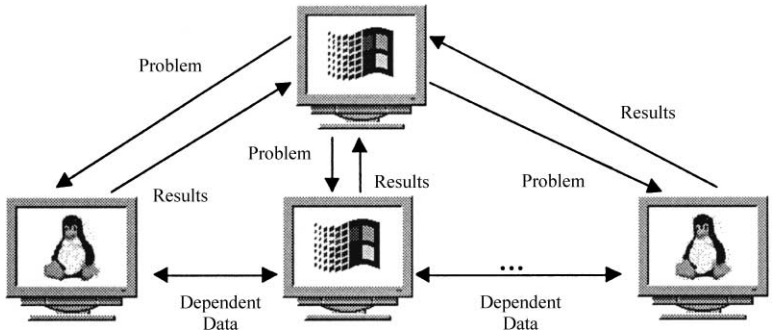

Fig. 4. The structure of the MPI–Delphi visual environment.

C PROCESSES

```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
RowsAssign(size,myrank,&nRows);
CreateMatrix(&A,myrank,nRows);
CreateMatrix(&B,myrank,nRows); /*Auxiliary matrix*/
/* It receives configuration from Delphi*/
MPI_Bcast(confActual,7,MPI_FLOAT,0,
          MPI_COMM_WORLD);
InitMatrix(A,myrank,nRows,size,confActual);
InitMatrix(B,myrank,nRows,size,confActual);
AllConverged = 0;  NumIterations = 0;
While (!allConverged) {
 NumIterations++;
 DoIteration(A, B, nRows); /*Calculate temperatures*/
 AllConverged = Converge(A,B,nRows, confActual[6]);
 ComunicateSharedRowUp(myrank,A);
 ComunicateSharedRowDown(myrank,A);
 MPI_Allreduce(&allConverged,&convAux, 1,
          MPI_INT,MPI_LAND,  MPI_COMM_WORLD);
 AllConverged = convAux;
/*It sends submatrix calculated to Delphi in order to print*/
 SendMatrix(A,nRows);

}
FreeMatrix(&B,nRows);
FreeMatrix(&A,nRows);
MPI_Finalize();
```

DELPHI PROCESS

```
MPI_Init;
MPI_Comm_rank(@myrank); (*myrank=0 for Delphi process*)
MPI_Comm_size(@size);
Matrix0 := FullMatrix.Create;
CreateConfiguration(confActual);
(* confActual[6] == ConvergenceLimit*)
MPI_Bcast(@confActual,7,MPI_FLOAT,0);
NumIterations := 0;
InitPicture;
While (allConverged = 0) do Begin
  Inc(numIterations);
  AllConverged:= 1;
  MPI_Allreduce(@allConverged,@ConvAux,
                  1, MPI_INT,MPI_LAND);
  AllConverged:= ConvAux;
  Matrix0.ReceiveMatrices;
  PrintPicture(Matrix0);
End;
Matrix0.Free;
MPI_Finalize;
```

Fig. 5. The parallel Jacobi relaxation algorithm coded in our MPI–Delphi interface.

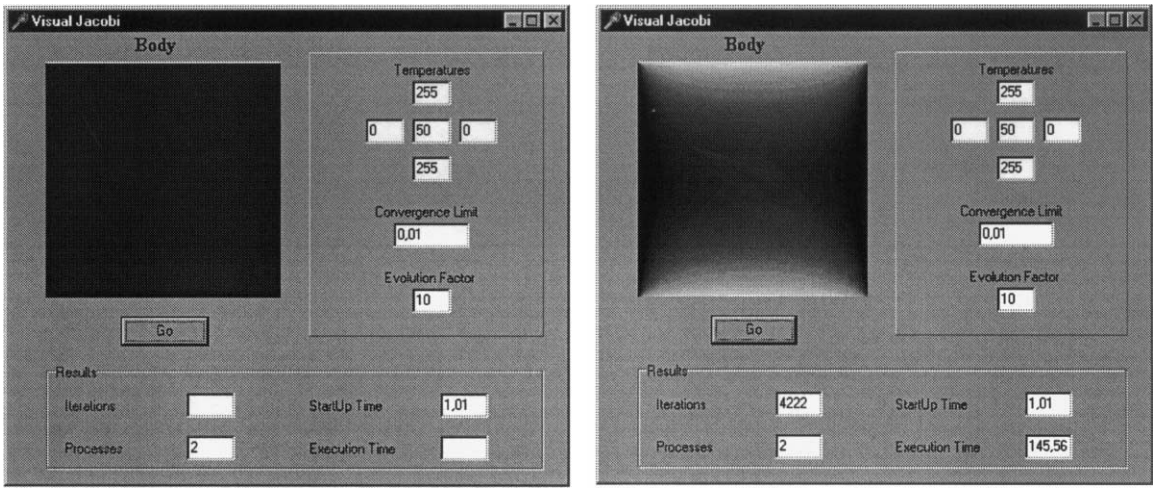

Fig. 6. Initial (left) and final (right) pictures of the Visual Jacobi relaxation algorithm.

iteration of the algorithm when implemented in parallel. Moreover, the user can easily introduce the values of the temperatures on the four boundaries, the initial temperature of the body and the convergence factor.

One problem that we found with this application was the fact that in order to observe the evolution in each iteration, each calculation process (Linux or NT) must send its portion of the body to the graphical Windows process. Doing this would need a great amount of data communication in every iteration, with the result that the parallel version is not scalable (e.g. when the number of processors reaches a certain quantity, the parallel version performs worse than the sequential one).

To solve this problem, our parallel program periodically displays the temperature of each internal point of the body. The period is user-defined, and we have named it as the λ parameter. It represents the number of iterations between two successive displays of intermediate results. If we want to see the partial results more frequently, then we will have to reduce the value of λ. If, on the other hand, we want the simulation to operate faster, then we can increase this value, but then the graphical animation will contain fewer frames (intermediate images). Several λ values were used in our simulations, in order to determine the value of this parameter which gives both good user visualization of the evolution of temperature, and optimum performance. Fig. 6 illustrates the first and last windows of the application. Higher values of the temperature are displayed with brighter light intensity values.

4. Using MPI–Delphi in a scientific application: Visual P-EDR

In this section, we show the use of the MPI–Delphi environment for a scientific, data-intensive application: the P-EDR algorithm. We first give details of the foundations of this algorithm. Then, we describe how the P-EDR algorithm was implemented, and how we divided the work between the two kinds of processes comprising the application (one graphical Windows process and several calculation processes). This section ends with a performance analysis of our algorithm for a cluster of PCs.

4.1. Foundations of the algorithm

EDR is a new iterative algorithm for non-parametric density estimation [25]. It was designed to solve a classic statistical problem, that of density estimation from samples, but with an important extension: the treatment of uncertainty. With this algorithm, traditional kernel density estimation methods are extended to accept uncertain observations modeled by likelihood functions. A variable kernel approximation scheme is derived, where the locations and widths of the components are obtained from the likelihood functions of the samples in an iterative procedure. Our algorithm can be considered as an improvement of the classic rectification method proposed by Lucy [17] using a well-known regularization tool, Parzen's method.

P-EDR (which stands for parallel empirical deconvolution by regularization) constitutes the parallel implementation of the EDR algorithm. It was designed following the SPMD model (e.g. single program, multiple data) in which a single user process runs on each processor in the system, using different data style [15]. Distributed resources are optimally exploited, making the system capable of estimating densities from large databases of uncertain samples. The aim is an efficient treatment of the distribution of this structure among processors, together with a low communication cost scheme, in order to obtain a highly scalable parallel algorithm.

Initially, P-EDR uses a database of N elements, which represents the original means and deviations (the N input samples with uncertainty). Then, the algorithm generates, after the necessary iterations, N means and deviations (referred as μ and σ). From these values, the new density estimation function ($p(x)$) can be calculated.

Fig. 7 summarizes the data flow in the P-EDR implementation.

At each iteration the new values of means and deviations are recalculated. In order to obtain these values, a table of $N \times N$ elements must be filled in. This table stores the temporal values participating in the calculation of such means and deviations. This algorithm structure is ideal to take advantage of parallelism. We parallelized the algorithm in the following manner: means and deviations can be calculated without the need for communications by distributing columns of this table among the processes. Each

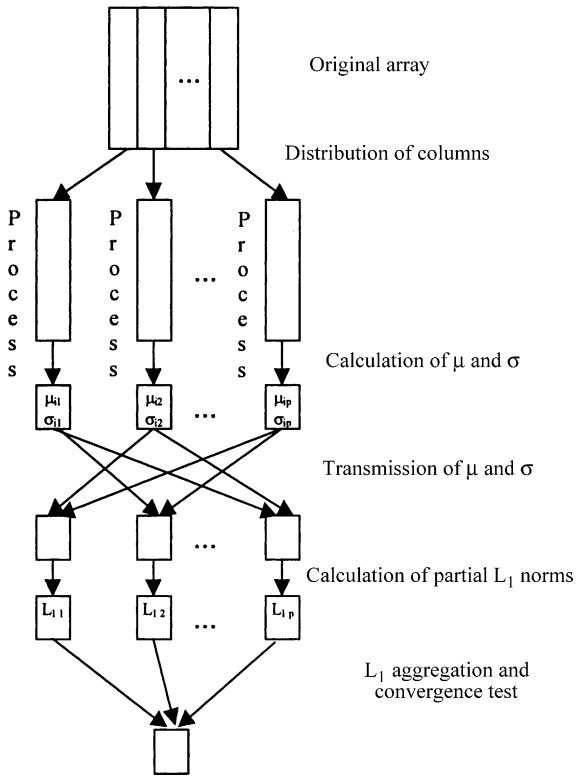

Fig. 7. Data flow in P-EDR implementation.

process is assigned a set of columns of the table, and it has to obtain the values of means and deviations for these columns. However, some communications are needed, due to the fact that every process must know the values of means and deviations worked out by the rest of processes from the previous iteration, in order to compute the new ones, and also to carry out the convergence test. Therefore, when a process obtains its local values, it must transmit them to the rest of the processes. Here, there is a first synchronization point of the algorithm.

Furthermore, the convergence test can also take advantage of parallelism. This test is based on the calculus of the L_1 norm, the value that measures the difference between two successive estimations of $p(x)$. It is obtained by splitting a sufficiently wide interval (which captures the whole set of examples, with some margin at the extremes) into I small subintervals, and subsequently performing a numerical integration by the trapezoids method. Thus, these subintervals can be distributed between all the processes, so each of them calculates a local value of the L_1 norm. Nevertheless, calculation of the L_1 norm requires means and deviations to be obtained, because these values are obviously used in the estimate of this rule. Moreover, the parallel calculus of the L_1 norm must be initiated only when every process has obtained the values of means and deviations corresponding to the columns of the table assigned to it, and it has sent these to the rest of processes. Once each process has computed the numerical value of the integral in the subintervals assigned to it, the global L_1 norm is obtained as the sum of the local values calculated by every process. Therefore, there is here a second point of synchronization.

4.2. The implementation using MPI–Delphi

Visual P-EDR is an implementation of P-EDR with a graphical interface used to show how the algorithm progressively converges within each iteration, and to supervise when the solution has reached a fixed (stable) point.

Fig. 8 shows the two parts of the application code, both using MPI, in the same way as we have done before. On the left, written in C language, the main code to implement the P-EDR algorithm, in a Linux/UNIX/NT environment using the standard MPI API for C language; on the right, written in Pascal language, the code of the process which initiates the parallel execution and shows the results graphically.

One interesting use of this graphical version was to refine the EDR algorithm's heuristics quickly in the development stage. With the help of the graphical interface, we could see when we obtained a good solution, in order to establish the stop condition. This possibility saved us a considerable amount of time (due to the parallel nature of the implementation) and it represents one of the most useful capabilities of MPI–Delphi.

Fig. 9 illustrates the first and last windows of the application. True (original unknown density) and P-EDRs estimated probability densities are drawn. As we show in this picture, the proposed method was able to efficiently recover the true density of moderately degraded data at a remarkably fast convergence rate. The next section introduces some execution times obtained from a performance evaluation. We include this information to justify the benefits that

C PROCESSES

```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Bcast(cfg, 4, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Bcast(original_means, n_samples, MPI_FLOAT, 0,
                        MPI_COMM_WORLD);
MPI_Bcast(original_desv, n_samples, MPI_FLOAT, 0,
                        MPI_COMM_WORLD);
Initiate_Aux_Table(&tabla);
Initiate_Aux_Prob(&prob);
Initiate_Means(&calculated_means, original_means, n_samples);
Initiate_ Desv(&calculated_desv, conf.h, n_samples);
do {
  Calculate_Local_Table;
  Calculate_Local_Means;   Calculate_Local_Desv;
  For (i=1;i<=numprocesses;i++) {
    If (myrank==i) {    (* Send local means and deviations *)
      MPI_Bcast(calculated_means +offset, size, MPI_FLOAT, i,
                        MPI_COMM_WORLD);
      MPI_Bcast(calculated_desv+offset, size, MPI_FLOAT, i,
                        MPI_COMM_WORLD);
    } else {
      MPI_Bcast(calculated_means+offset, k, MPI_FLOAT, i,
                        MPI_COMM_WORLD);
      MPI_Bcast(calculated_desv+offset, k, MPI_FLOAT, i,
                        MPI_COMM_WORLD);
    }
  }
  total_convergence = 0;
  Calculate_Local_Convergence(&local_conv);
  MPI_Allreduce(&local_conv, &total_convergence ,1,
          MPI_FLOAT,MPI_SUM,MPI_COMM_WORLD);
} while (total_convergence > configuration.norma);
MPI_Finalize();
```

DELPHI PROCESS

```
MPI_Init;
MPI_Comm_rank(@myrank);  (*myrank=0 for Delphi process*)
MPI_Comm_size(@size);
Read_Input_File;    (* Contains original means, deviations and
                       some other configuration parameters *)
MPI_Bcast(configuration, 4, MPI_FLOAT, 0);
MPI_Bcast(original_means, n_samples, MPI_FLOAT, 0);
MPI_Bcast(original_desv, n_samples, MPI_FLOAT, 0);
Repeat
  for i:=1 to n_processes do begin    (*Receive from calculation
                                          processes *)
    MPI_Bcast(calculated_means+offset, ncolumnas,
                        MPI_FLOAT, i);
    MPI_Bcast(calculated_desv +offset, ncolumnas,
                        MPI_FLOAT, i);
  end;
  PrintPicture(New_density);
  constant_c = 0;
  MPI_Allreduce(&constant_c, &converg, 1, MPI_FLOAT,
                                          MPI_SUM);
  Inc(num_iteraciones);
Until (converg <= configuration.norma);
Write_Output_File(calculated_means, calculated_desv);
MPI_Finalize;
```

Fig. 8. Visual P-EDR algorithm coded in our MPI–Delphi interface.

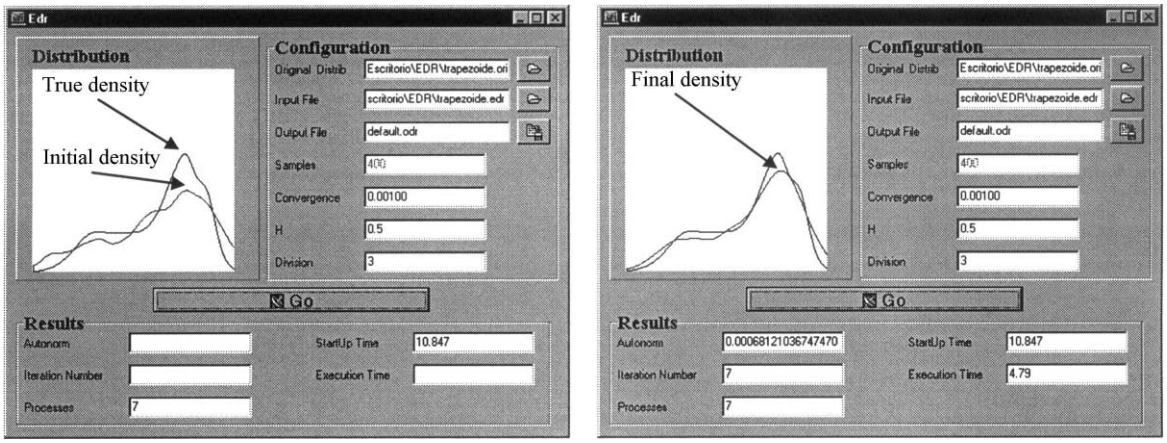

Fig. 9. Initial (left) and final (right) pictures of the Visual P-EDR algorithm.

we have obtained by debugging the EDR algorithm's heuristics using Visual P-EDR.

The details of the parallel implementation are described in [16] and an exhaustive performance analysis of P-EDR can be found in [1]. This point is beyond the scope of this paper. We merely wish to show how MPI–Delphi has been used in the solution of a real and computationally expensive problem, in which graphical user interaction was needed.

4.3. The MPI–Delphi performance analysis

At the time of writing this paper, our research group had a cluster of PCs with a mixture of Linux and Windows NT operating systems. We carried out these experiments in a cluster of Intel Pentium 200 MHz processors with 32 MB main memory and 256 KB L2 cache memory. The communication between the processes was achieved using a Fast Ethernet local area network. We used a Fast Ethernet 3Com 905-network adapter as the communication channel. In this computer configuration, we noted that Windows sockets are slower than TCP/IP sockets on Linux [24,26]. Therefore, we used a homogeneous cluster of Linux computers in this performance evaluation. Windows NT Server v.4.0 was used as the operating system in the graphical Windows processor. The rest of them were equipped with Red Hat Linux 5.0, which provides the Linux 2.0.32 kernel version. However, in other experiments we have carried out, the heterogeneous configuration was used with similar results.

As we stated above, the Windows' side of the application has been developed using Delphi plus MPI–Delphi. Specifically, we used the Delphi 3.0 programming environment. C language was used when developing the Linux components of the application, and these components were compiled using the GNU GCC compiler.

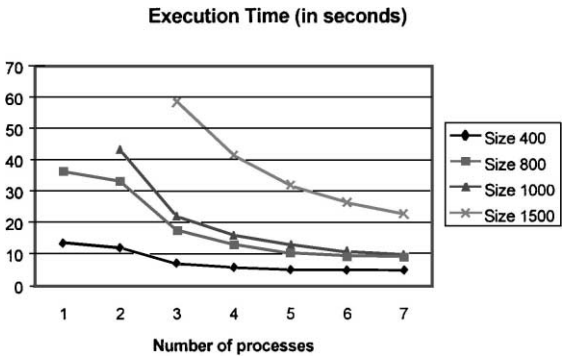

Fig. 10. Execution time.

Finally, it is important to note the compiler options related to code optimizations. They constitute an important factor which must be taken into consideration in order to obtain good results. We have compiled Delphi program with the best optimization options provided by the IDE. C programs were compiled using the GCC compiler with the –O2 optimization option.

Visual P-EDR was tested using samples of 400, 800, 1000 and 1500 unidimensional items, and adequate values of h and the accuracy factor for convergence. Table 1 summarizes the results obtained with each sample size and using different numbers of processors (one process per processor). Fig. 10 also shows these results graphically.

We can observe that a sequential version of EDR algorithm was not executed with samples of 1000 or 1500 items. In these cases more than 32 MB of RAM were needed to solve the problem, thus involving a large amount of swapping. This fact illustrates one of the main advantages of clusters of workstations: the

Table 1
Execution times of Visual P-EDR

Processes	Size 400		Size 800		Size 1000	Size 1500
	Time (s)	Speed-up	Time (s)	Speed-up	Time (s)	Time (s)
Sequential	13.50	1	36.27	1	–	–
2	13.50	1	33.11	1.09	43.00	–
3	6.96	1.94	17.41	2.08	21.87	58.40
4	5.73	2.36	12.86	2.82	15.87	41.36
5	5.00	2.70	10.31	3.52	12.89	31.77
6	5.00	2.70	9.42	3.85	10.80	26.24
7	4.79	2.81	9.20	3.94	9.63	22.68

Table 2
Execution times of non-Visual and Visual P-EDR

Processes	Time (size 1000, non-visual P-EDR) (s)	Time (size 1000, visual P-EDR) (s)
3	20.80	21.87
4	15.69	15.87
5	12.95	12.89
6	11.16	10.80
7	9.53	9.63

exploitation of distributed-memory resources (RAM). Though, it is not possible to obtain the speed-up for these cases, we can appreciate that time employed to solve the problem with seven processors is less than half the time needed to do it with three processors, indicating very good scalability in the implementation.

Evidently, an interface for parallel programming offers low performance levels. Thus, in order to be fair, it is necessary to give some performance results to evaluate the overheads introduced by our interface related to MPICH 1.0.13. This performance analysis was accomplished by comparing Visual P-EDR execution times with the running times of the P-EDR algorithm written in C using the MPICH standard library, in order to evaluate the overhead introduced by MPI–Delphi.

In order to evaluate the overheads introduced by MPI–Delphi, we have implemented a pure C version of P-EDR. We have executed this non-Visual P-EDR implementation with the 1000 element sample. Table 2 shows the execution times obtained for the non-visual and visual version of P-EDR. It can be seen that the overhead introduced by our MPI–Delphi implementation is very low (only 1 s in the worst case). Thus, we pay a very low price for the visualization of the results using MPI–Delphi.

5. Related work

Visualization has played a role in parallel and distributed computing for many years, from paper-based diagrams for understanding parallel computation, to program animation and visual debugging aids [8,13].

Recently, several tools have been created to improve software development in high-performance computing. There are some programming environments for developing, porting and tuning of parallel applications, such as [14]. Wismüller [29] describes a set of integrated tools the aim of which is to write parallel programs efficiently; Maier and Stellner [19] describes another tool to manage the distributed resources in a heterogeneous NOW. Similarly, off-line tools for performance analysis (monitoring, trace data, visualization) in NOWs have appeared, as is described in [6]. Several visual parallel programming languages also exist, such as CODE [22] and HENCE. These languages are more oriented to the representation of parallel programs as a directed graph, in which nodes with certain icons represent sequential computations and the graph as a whole represents the parallel structure of the program. These research projects are not directly related to our work, as we have centered on the user interface, but the basic goal is the same: a simple way of developing parallel programs.

The MPI standard has been implemented in a great variety of systems and languages. We would like to mention some varieties of MPI:

- OOMPI [18] and MPOOL [9] provide object-oriented definitions of MPI. Due to the fact that Object Pascal is an object-oriented programming language, we could have adopted one of these specifications in order to provide message-passing functionality to Delphi. However, we preferred to implement MPI–Delphi following the MPI standard.
- mpiJava is one of the implementations of MPI for Java programming environments now available [4]. This work was undertaken in order to offer MPI functionality to one of the most popular programming languages, Java, but adopting an object-oriented methodology.

At present, there are some implementations of MPI for the Windows NT/2000 operating system. Our work was based on one of them, specifically [20], as this version is frequently updated. There are also other

implementations of MPI for Windows versions [5] [21]. We have heard no reports of any other implementation of MPI standard for the Delphi programming environment.

6. Conclusions

With the increasing popularity of distributed systems as a cost-effective means for high-performance computing, efficient and portable visual programming interfaces become increasingly important. Our work has focused on offering the possibility of easily programming parallel applications with a graphical interface. In such environments, programming is simplified by providing a set of standard components, which are adjusted by the programmer to his/her particular application. In this way, the creation of a user interface is a simple task, using graphical components provided by Delphi. Thus, the programmer saves time and effort when implementing a parallel algorithm. The Delphi visual programming environment provides this capability, but the main problem is related to the possibility of creating parallel applications within this environment, given that, until now, there has been no implementation of standard MPI parallel programming libraries for Delphi.

MPI–Delphi constitutes, therefore, the first approach for Delphi programmers to parallel programming within a message-passing paradigm. Extending the parallel programming to a visual programming environment entails many advantages, such as high level debugging tools or automatic creation of a user interface. These additional advantages are achieved at very low cost, as was shown in Section 5, when comparing execution times of Visual P-EDR with a pure C and MPICH implementation of P-EDR.

The MPI–Delphi interface is suitable for some specific kinds of problems, such as monitoring parallel programs of long execution time, or computationally intensive graphical simulations. In addition, MPI–Delphi has shown itself to be a good tool for research, as the development of new algorithms can be carried out quickly and, therefore, time spent on the debugging of such algorithms is reduced. The P-EDR algorithm constitutes a good example of this last affirmation.

The inter-machine communication software provided by standard workstation operating systems, such as TCP/IP, incurs an overhead that is at least one order of magnitude larger than the communication overheads on multicomputers. This performance difference is mainly attributed to the generality of communication behavior that a generic operating system is required to support. There are several implementations which seek to reduce this latency [28]. As part of our future work, we will try to apply some of these ideas to our MPI–Delphi implementation.

Currently, we are working on the application of the MPI–Delphi interface in several fields, such as medium- and high-level computer vision, memory and calculus intensive machine learning algorithm, and several classic statistical problems.

[5] In the NHSE (National HPCC Software Exchange) there are available several implementations (http://www.nhse.org).

Acknowledgements

This work has been partially supported by the Spanish CICYT under Grant TIC97-0897-C04-03. Finally, we would like to thank Alberto Ruiz for his collaboration on the foundations of the P-EDR algorithm.

References

[1] M. Acacio, J.M. García, P.E. López-de-Teruel, A performance evaluation of P-EDR in different parallel environments, in: Proceedings of the PDPTA'99, CSREA Press, 1999, pp. 744–750.

[2] M. Acacio, J.M. García, P.E. López-de-Teruel, O. Cánovas, The MPI–Delphi interface: a visual programming environment for clusters of workstations, in: Proceedings of the PDPTA'99, CSREA Press, 1999, pp. 1730–1736.

[3] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, W. Zwaenepoel, TreadMarks: shared memory computing on networks of workstations, IEEE Computer 29 (2) (1996) 18–28.

[4] M. Baker, B. Carpenter, G. Fox, S.H. Ko, S. Lim, mpiJava: an object-oriented Java interface to MPI, in: Proceedings of the 1999 Second Merged IPPS/SPDP, IEEE Computer Society Press, 1999.

[5] M.A. Baker, G.C. Fox, MPI on NT: a preliminary evaluation of the available environments, in: Proceedings of the 1998 First Merged IPPS/SPDP, Lecture Notes in Computer Science, Springer, Berlin, 1998, pp. 549–563.

[6] M. Bubak, W. Funika, J. Moscinski, Tuning the performance of parallel programs on NOWs using performance analysis

tool, in: Proceedings of the 1998 Applied Parallel Computing, Lecture Notes in Computer Science, Vol. 1541, Springer, Berlin, 1998, pp. 43–47.

[7] R. Buyya, High Performance Cluster Computing, Vol. 2, Prentice-Hall, Englewood Cliffs, NJ, 1999.

[8] K.C. Cox, G.-C. Roman, Visualising concurrent computations, in: Proceedings of the 1991 IEEE Workshop on Visual Languages, 1991, pp. 18–24.

[9] G. Destri, A. Poggi, An object-oriented extension to MPI, in: Proceedings of the AICA'96, 1996, pp. 187–202.

[10] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message Passing Interface, MIT Press, Cambridge, MA, 1994.

[11] W. Hu, W. Shi, Z. Tang, JIAJIA: an SVM system based on a new cache coherence protocol, in: Proceedings of the 1999 High Performance Computing and Networking (HPCN'99), Lecture Notes in Computer Science, Springer, Berlin, 1999, pp. 463–472.

[12] A. Itzkovitz, A. Schuster, MultiView and Millipage — fine-grain sharing in page-based DSMs, in: Proceedings of the 1999 Third Symposium on Operating Systems Design and Implementation (OSDI'99), 1999, pp. 215–228.

[13] H. Koike, T. Takada, T. Masui, VisualLinda: a framework for visualizing parallel Linda programs, in: Proceedings of the 1997 IEEE Visual Languages Symposium, 1997, pp. 174–180.

[14] W. Krotz-Vogel, H.C. Hoppe, The PALLAS parallel programming environment, in: Proceedings of the 1997 Fourth Euro PVM/MPI Meeting, Lecture Notes in Computer Science, Springer, Berlin, 1997, pp. 17–24.

[15] B. Lester, The Art of Parallel Programming, Prentice-Hall, Englewood Cliffs, NJ, 1993.

[16] P.E. López-de-Teruel, J.M. García, M. Acacio, O. Cánovas, P-EDR: an algorithm for parallel implementation of Parzen density estimation from uncertain observations, in: Proceedings of the 1999 Second Merged IPPS/SPDP, IEEE Computer Society Press, 1999, pp. 563–568.

[17] L.B. Lucy, An iterative technique for the rectification of the observed distributions, Astron. J. 79 (6) (1974) 745–754.

[18] A. Lumsdaine, J.M. Squyres, B.C. McCandless, Object oriented MPI: a class library for the message passing interface, in: Proceedings of the POOMA'96, February 1996.

[19] U. Maier, G. Stellner, Distributed resource management for parallel applications in networks of workstations, in: Proceedings of the HPCN Europe, 1997, Lecture Notes in Computer Science, Springer, Berlin, pp. 462–471.

[20] J. Marinho, J.G. Silva, WMPI — message passing interface for Win32 clusters, in: Proceedings of the Fifth Euro PVM/MPI, Lecture Notes in Computer Science, Springer, Berlin, 1998, pp. 113–121.

[21] J. Meyer, H. El-Rewini, A. Helal, WinMPI: an MPI port to Windows, MS Thesis, Department of Computer Science, University of Nebraska, Omaha, July 1995.

[22] P. Newton, J.C. Browne, The CODE 2.0 Graphical parallel programming language, in: Proceedings of the 1992 ACM International Conference on Supercomputing, 1992, pp. 167–177.

[23] G.F. Pfister, In Search of Clusters: the Ongoing Battle in Lowly Parallel Computing, Prentice-Hall, Englewood Cliffs, NJ, 1998.

[24] J. Piernas, A. Flores, J.M. García, Analyzing the performance of MPI in a cluster of workstations based on fast Ethernet, in: Proceedings of the 1997 Fourth Euro PVM/MPI Meeting, Lecture Notes in Computer Science, Springer, Berlin, 1997, pp. 17–24.

[25] A. Ruiz, P.E. López-de-Teruel, M.C. Garrido, Kernel density estimation from heterogeneous indirect observation, in: Proceedings of the Learning'98, September 1998.

[26] P.S. Souza, L.J. Senger, M.J. Santana, R.C. Santana, Evaluating personal high performance computing with PVM on Windows and Linux Environments, in: Proceedings of the 1997 Fourth Euro PVM/MPI Meeting, Lecture Notes in Computer Science, Springer, Berlin, 1997, pp. 49–53.

[27] E. Speight, J.K. Bennett, Brazos: a third generation DSM system, in: Proceedings of the 1997 USENIX Windows/NT Workshop, 1997, pp. 95–106.

[28] M. Verma, T. Chiueh, An efficient communication system for parallel computing on network of workstations, in: Proceedings of the 1998 International Parallel Processing Symposium, PC-NOW'98 Workshop, 1998.

[29] R. Wismüller, T. Ludwig, A. Bode, R. Borgeest, S. Lamberts, M. Oberhuber, C. Röder, G. Stellner, The tool-set project: towards an integrated tool environment for parallel programming, in: Proceedings of the 1997 Second Sino–German Workshop APPT'97, 1997, pp. 9–16.

M. Acacio was born in Murcia, Spain on 30 October 1975. He is currently a PhD student and an Assistant Professor in the Engineering and Computer Technology Department at the University of Murcia (Spain). He received his MS degree in computer science in 1998 from the University of Murcia. His teaching includes several courses on computer architecture and real-time systems. As part of his research, he is working on cache coherence protocols and new directory organizations for cc-NUMA machines. His other areas of interest include cluster computing, interconnection networks and compilers.

O. Cánovas was born in Barcelona, Spain on 8 September 1975. He received his MS degree in computer science from the University of Murcia (Spain) in 1998, where he is currently pursuing a PhD in computer science. At present, he is a member of the Engineering and Computer Technology Department, where he is an Assistant Professor from 1999. His teaching includes courses on computer networks, and computer architecture. His current research interests include cluster computing, and development of security infrastructures for communication environments, such as public key infrastructures, credential-based systems and e-commerce scenarios.

J.M. García was born in Valencia, Spain on 9 January 1962. He received the Ingeniero Industrial (electrical engineering) and the PhD degrees from the Universidad Politecnica de Valencia (Valencia, Spain) in 1987 and 1991, respectively. In 1987, he joined the Departamento de Informatica at the Universidad de Castilla-La Mancha at the Campus of Albacete. From 1987 to 1993, he was an Assistant Professor of Computer Architecture and Technology. In 1994, he became an Associate Professor at the Universidad de Murcia (Spain). From 1995 to 1997, he served as Vice-Dean of the School of Computer Science (Facultad de Informatica). At present, he is the Director of the Computer Engineering and Technology Department (Departamento de Ingenieria y Tecnologia de Computadores), and also the Head of the Parallel Computing and Architecture Research Group. He has developed several courses on Computer Structure, Peripheral Devices, Computer Architecture and Multicomputer Design. His current research interests include Interconnection Networks and Cache-coherent Multiprocessors, File Systems, Cluster Computing and its Applications, and Video Compression. He has published over 30 refereed papers in different Journals and Conferences in these fields. Dr Garcia is a member of several international associations (IEEE Computer Society, ACM, USENIX), and also a member of some European associations (Euromicro and ATI).

P.E. López-de-Teruel was born in Lorca (Murcia, Spain) on 31 December 1970. He received his computer science and research adequacy degrees from the University of Murcia (Spain) in 1993 and 1999, respectively, where he is currently pursuing a PhD in Computer Science. In 1995, he joined the Computer Science and Systems Department at the University of Murcia, where he was an Assistant Professor until year 1997. From 1996 to 1997, he also served as a Secretary of that Department. At present, he is a member of the Engineering and Computer Technology Department, where he is an Assistant Professor from 1997. His teaching includes several courses on computer structure, digital systems, and computer architecture. His current research interests include cluster computing, and development and evaluation of parallel algorithms in different application areas, such as computer vision, machine learning, pattern matching and robot navigation. He has published over 10 refereed papers in several journals and conferences related to these fields.