The MPI-Delphi Interface: A Visual Programming Environment for Cluster of Workstations[*]

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

Dpto. de Ingeniería y Tecnología de Computadores
University of Murcia
Campus de Espinardo, s/n 30080 Murcia (Spain)
{meacacio, pedroe, jmgarcia}@ditec.um.es
ocanovas@fcu.um.es

Abstract. The goal of a parallel program is stated as *to reduce the execution time regarding the fastest sequential program solving the same problem*. Parallel programming is growing due to the widespread use of network of workstations in high performance computing. MPI has become the standard for implementing message-based parallel programs in distributed-memory computing environments. On the other side, visual programming environments try to make easier the task of developing applications. Delphi constitutes one of the most popular visual programming environments nowadays in the Windows operating system environment. In this paper we present MPI-Delphi, an implementation of MPI for writing parallel applications using Delphi. We explain how MPI-Delphi has been derived, how it has been used in a cluster of workstations with a mixed of Windows and Linux operating systems, and what MPI-Delphi is suitable for.

1. Introduction

In recent years, parallel distributed systems and parallel programming is doing a growing use of message passing paradigm for solving complex computational problems. Currently, networks of workstations (NOWs) are being considered as a good alternative to the use of expensive, dedicated high-performance systems. In fact, the increasing performance and availability of general-purpose microprocessors has fostered the spread of NOWs, either with shared or distributed memory, as an alternative to customized massively parallel systems (MPPs). So, a cluster of powerful PCs interconnected with a fast network could be seen as a reasonable and cost-effective alternative approach [1,10] for exploding parallelism.
Traditionally, Unix/Linux has been used as operating system in such clusters of PCs. However, Windows 95/98/NT is perhaps the most widely used operating system nowadays. Moreover, the Intel Pentium II (and III now), together with its Windows 98/NT operating system (the Wintel model), provides enough overall capacity to displace also the RISC/UNIX workstation in the engineering and scientific marketplace. This fact causes that each time more and more clusters of PCs exist with Windows 95/98/NT as operating systems. These operating systems have the advantage over Linux on the number of developed applications for them and on the ease of use that it offers. Among these applications, programming environments are not an exception. There is a great variety of programming environments available for Windows. Visual programming environments constitute one of the most interesting developing tools, because they make the creation of applications easier to the programmer.
With the advent of the MPI [4] standard, parallel programming using the message-passing style has reached 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 the last years a large variety of this kind of tools has 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. In addition, parallelism in a cluster of workstations with Windows 95/98/NT is possible, as some MPI and PVM versions for Windows have been implemented [9, 13].
Recently, several tools have been developed to improve the software development in high-performance computing. In [14] is described a set of integrated tools destined to write parallel programs efficiently; [8] describes another tool to manage the distributed resources in an heterogeneous NOW. Similarly, off-line tools for performance analysis (monitoring, trace data, visualization) in NOWs have appeared, as it is described in [3].

[*] This work has been partially supported by the Spanish CICYT under grant TIC97-0897-C04-03

There also exist several visual parallel programming languages, such as HENCE 2.0 [2]. These languages are more oriented to the representation of parallel programs like 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. However, our approach is more based on the visual programming of the user interface.

Nowadays, therefore, most programming projects are developed under visual environments. Delphi* 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, we are very interested in implementing parallel applications with Delphi, in order to take advantage of all these programming facilities.

The first and most obvious problem that we will find is that this environment uses Object Pascal (an object-oriented version of Pascal) as its native programming language, while available Windows MPI/PVM versions are accessible just from Visual C++# and Borland C++*.

Our work has centered in offering MPI functionality to this popular environment using one of these existing Windows MPI implementations. So, it will be possible to develop parallel applications using Delphi and making good use of the facilities that it supplies for applications with user interaction and/or complicated graphical requirements.

It is an accepted fact that using Windows 95/98/NT the performance of parallel applications often decreases with respect to workstations running UNIX/Linux. As a possible solution, we also propose an execution environment that takes advantage of the benefits of both Unix/Linux and Windows operating systems.

We have structured this paper as follows: first, we introduce how MPI-Delphi has been created (what problems we have found in its creation and how they have been solved), and then we describe the execution environment that we propose. Next, we show two sample applications created using MPI-Delphi, along with the visual results obtained for them. Finally, we expose the conclusions of the work.

2. The MPI-Delphi Visual Environment

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 had to be introduced. This section describes these modifications and the way we carried out the implementation.

Currently, several MPI implementations for Windows can be found. The main problem we have found is that the language they have been created for is C, and the environments they can be used from, Visual C++ or Borland C++.

We have used W32MPI v.0.9b [9] as the basis to develop a mechanism to provide MPI functionality to Delphi. This implementation is freely available at the web pages of Coimbra University. W32MPI v.0.9b is a full MPI standard implementation for Microsoft Win32 platforms. It is completely compatible with MPICH 1.0.13 (an also freely MPI implementation of 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 bits platforms) is accomplished through a common TCP/IP network.

2.1 Some important aspects of the implementation of MPI-Delphi

In the implementation of MPI-Delphi, we had two main objectives:

1. To carry out an implementation in which the functionality of W32MPI were offered to Delphi in an easy way. Delphi's programmers should make use of the MPI functionality in a simple, transparent way, without having to know the internal details of the implementation. Users should only know how to use the MPI standard functions.
2. To carry out the necessary adaptation between C or Fortran and Pascal for the MPI functions, because MPI has been defined for C and Fortran languages.

In order to cope the first objective, two solutions were analyzed:

* Delphi and Borland C++ are a trademark of Borland Corporation Inc.

Visual C++ is a trademark of Microsoft.

1. The use of DDE (Dynamic Data Exchange) to communicate W32MPI and Delphi. It would involve a server application written in C++, offering MPI functionality to client applications written in Delphi. This server application should run before Delphi's parallel programs.
2. Creating a DLL (Dynamic Link Library) providing MPI functionality. This option also means using C++ in order to create the DLL.

In both cases, a DCU (Delphi Compiled Unit) has to be used to access to the MPI functionality.
Our final choice is based in terms of viability, scalability and facility. Thus, we concluded that option 2 represented the most suitable solution, because the programmer would simply use the functions contained in the DCU just the way he would do when using any other DCU. Figure 2 shows the elements implicated in the final solution.

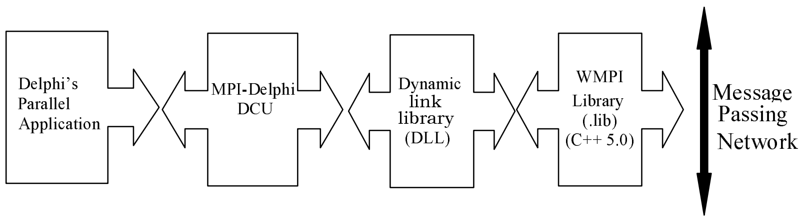

Delphi's Parallel Application
MPI-Delphi DCU
Dynamic link library (DLL)
WMPI Library (.lib) (C++ 5.0)
Message Passing Network

Figure 1. Relation between elements of MPI-Delphi.

The other objective was related to the MPI specification. C programming language could be considered more near to Pascal than FORTRAN, so we used MPI specification for C as our basis, but introducing two small modifications:

1. MPI_Init (int argc, char *argv): the arguments *argc* and *argv* are passed directly inside the DCU, so we free the Delphi's programmer of this task.
2. Some modifications on the data types. Many MPI-C functions return a value representing a boolean as an integer value (i.e. MPI_Test). In Object Pascal, boolean data type is available, so it was used as the returned type in this kind of functions.

Besides, communicators have been omitted in this first version of MPI-Delphi. The reason was that the first goal of this version was to determine the possibility of parallel programming in Delphi, more than to implement the whole functionality of the standard. Therefore, and in the sake of simplicity, the communicator argument was eliminated in the implemented functions in the DCU.

2.2 Our working environment

MPI-Delphi can be used in a cluster of workstations using Windows 95/98/NT in each PC of the cluster. But this involves a loss of speed performance in communications, as Windows sockets usually are much slower than TPC/IP sockets on Linux/UNIX environments [13].
W32MPI v.0.9b, and therefore MPI-Delphi, is compatible with MPICH v.1.0.13 for Linux/Unix, so it is possible to combine processes in execution in all these operating systems. We propose a solution for taking advantage of both aspects: speed performance of Linux/UNIX (in both communication and processing tasks), and programming facilities of Windows.
This working environment uses two kinds of processes:

1. A unique Windows process constituting the user interface, and that would contain all the graphical part of the parallel application. This process should be written using Delphi and MPI-Delphi.
2. Some Linux processes written in C language and used to effectuate the computations. They would have to send periodically results of their computations to the Windows process.

Note that this way of developing applications involves using two different programming languages: Object Pascal and C. However, observe that the proposals of both kind of processes are very different: The first one (Delphi programmed) has to manage the graphical interface and the user interaction, while the rest of processes (C programmed) manage the heaviest computational load (and therefore susceptible of parallelism) of the concrete algorithm implemented. So, it is even appropriate to develop them in different languages, each one suitable for each kind of task. Figure 2 shows the described working environment.

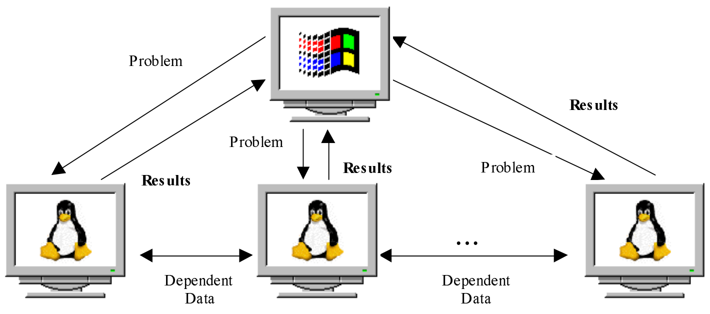


Figure 2. The working environment

3. Using MPI-Delphi

This section presents two examples of use of MPI-Delphi. Firstly, we show an easy example: the implementation of the well-known Jacobi's relaxation method, applied to solve the problem of the heat diffusion in a body [5]. We called this program Visual Jacobi, as it incorporates a nice 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. Secondly, we study the use of MPI-Delphi to develop a scientific application: the parallel visual version of the EDR algorithm [6], a new method for estimation of probability density functions from uncertain samples.

3.1 The visual Jacobi's relaxation method

Jacobi´s relaxation method is an iterative algorithm for solving differential equations. An example of the application of this algorithm is to consider a body represented by a two-dimensional array of particles. This body is in direct contact with a fixed value of temperature on the four boundaries; all four boundaries can have different temperatures, and each particle in the body has an initial value of temperature. The algorithm is solved by setting the temperature of each particle to be the mean of the temperatures of the four boundary particles. This calculus is carried out for each one of the particles, until a stability situation is reached.
Visual Jacobi implements the classical parallel Jacobi's relaxation algorithm (body division by rows, with these groups of rows being calculated by different processes), but, besides, it also shows graphically the evolution of the body. Figure 3 shows the two parts of the application code, both in MPI. On the left side, written in C language, the main code to calculate in parallel the Jacobi's relaxation algorithm, in a Unix environment. On the right side, written in Pascal language, the code of the process that starts the parallel execution and shows graphically the results.
The main advantage of the Delphi process is that it interacts with the user in a user-friendly fashion, allowing him to observe the evolution of the partial results obtained through iteration of the algorithm when implemented in parallel. In this way, we can debug the MPI parallel implementation easier. Moreover, the user can introduce very kindly the values of the temperatures on the four boundaries, the initial temperature of the body and the convergence factor. In several moments, the temperature of each internal point of the body will be displayed. To manage it, we introduced a new parameter, λ, that represents the number of iterations between two successive intermediate results displaying. This parameter was used to avoid the displaying of the temperatures of the body in each iteration, as doing it so would cause a great loss of performance: the iterative visualization of the results implies an additional communication to send to the visual process the values of the temperatures that the rest of the processes calculate. If we want to take a look at the partial results more frequently, then we will have to decrement the value of λ. If, on the other hand, we want the simulation to operate faster, then we can increment this value, but then the graphical animation will contain less frames (intermediate images). Figure 4 illustrates the first and last windows of the application. The higher values of the temperature, the brighter light intensity values were displayed.

C PROCESSES

```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
RowsAsign(size,myrank,&nRows);

createMatrix(&A,myrank,nRows);
createMatrix(&B,myrank,nRows);  /*Auxiliary matrix*/

MPI_Bcast(confActual,7,MPI_FLOAT,0,
            MPI_COMM_WORLD);     /* It receives
                          configuration from Delphi*/

initMatrix(A,myrank,nRows,size,confActual);
initMatrix(B,myrank,nRows,size,confActual);
allConverged = 0;  /*until all processes converge*/
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;
  sendMatrix(A,nRows);   /*It sends submatrix calculated
                          to Delphi in order to print*/

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

DELPHI PROCESS

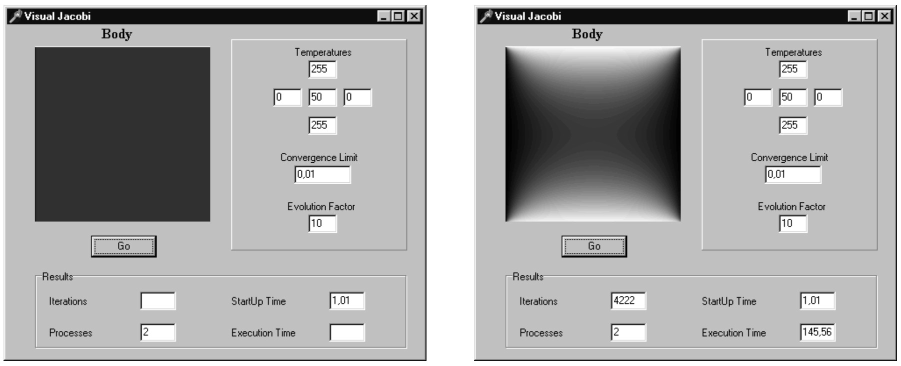
```
MPI_Init;
MPI_Comm_rank(@myrank);  (* myrank == 0 for the
                               Delphi process *)
MPI_Comm_size(@size);

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.ReceiveMatrixs;
  PrintPicture(Matrix0);
End;
Matrix0.Free;
MPI_Finalize;
```

Figure 3. The parallel Jacobi's relaxation algorithm coded in our MPI-Delphi interface

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

EDR algorithm is a recent result of our Research Group [12]. This algorithm has also an iterative nature, as the Jacobi method described before, though its theoretical foundations are very different.
Firstly, we are going to introduce the utility of EDR. It was designed to solve a classical 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 classical rectification method proposed by Lucy [7] using a well-known regularization tool, Parzen's method.
Visual P-EDR is an implementation in parallel of the EDR algorithm with a graphical interface, used to show how the algorithm converges more a more within each iteration, and to supervise when the solution has reached a fixed (stable) point. One interesting use of this graphical version was to refine the algorithm's heuristics quickly, with the help of the graphical interface (we could see when we obtain a right solution, in order to establish a stop condition).

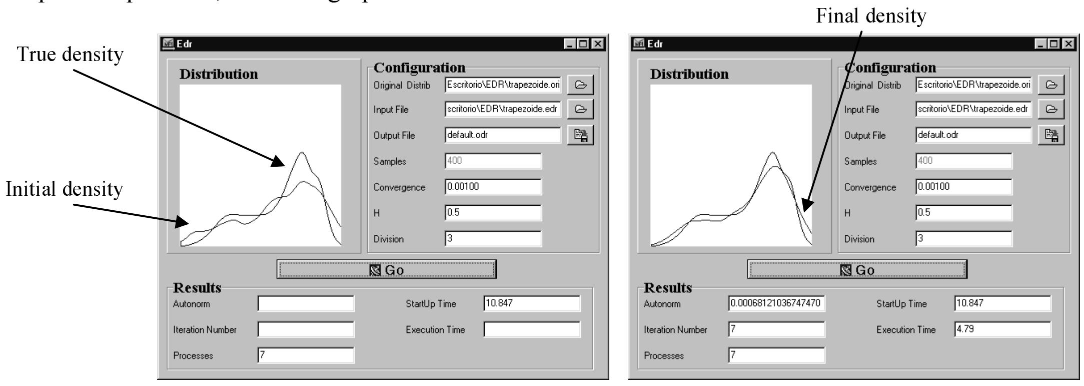


Temperatures: 255; 0, 50, 0; 255
Convergence Limit: 0,01
Evolution Factor: 10
Results
Iterations: | StartUp Time: 1,01
Processes: 2 | Execution Time:

Temperatures: 255; 0, 50, 0; 255
Convergence Limit: 0,01
Evolution Factor: 10
Results
Iterations: 4222 | StartUp Time: 1,01
Processes: 2 | Execution Time: 145,56

Figure 4. Initial (left) and final (right) windows of the visual Jacobi's algorithm

Figure 5 illustrates the first and last windows of the application. True (original unknown density) and P-EDR's estimated probability densities are showed. As we show in the figure, the proposed method was able to efficiently recover the true density of moderately degraded data with a remarkably fast convergence rate [6]. The details of the parallel implementation are exhaustively described in [6], but they are out of the scope of this article. Our interest is just 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.



Configuration
Original Distrib: Escritorio\EDR\trapezoide.ori
Input File: scritorio\EDR\trapezoide.edr
Output File: default.odr
Samples: 400
Convergence: 0.00100
H: 0.5
Division: 3
Results
Autonorm: | StartUp Time: 10.847
Iteration Number: | Execution Time:
Processes: 7

Configuration
Original Distrib: Escritorio\EDR\trapezoide.ori
Input File: scritorio\EDR\trapezoide.edr
Output File: default.odr
Samples: 400
Convergence: 0.00100
H: 0.5
Division: 3
Results
Autonorm: 0.00068121036747470 | StartUp Time: 10.847
Iteration Number: 7 | Execution Time: 4.79
Processes: 7

Figure 5. Initial (left) and final (right) windows of the visual P-EDR algorithm

3.3 Our hardware and software environment

Our research group has 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 cache memory. We used a Fast Ethernet 3Com 905-network adapter as the communication channel. Windows NT v.4.0 was used as the operating system in one of the PC's. The rest of them used Linux 2.0.32 operating system.
Delphi programs were compiled with the best optimization options provided by the IDE (Integrated Development Environment), while C programs were compiled using the GNU gcc compiler with –O2 optimization option.

4. 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 programming parallel applications with a graphical interface in a fast way. In this way, the programmer saves effort and time costs when implements a parallel algorithm. Delphi visual programming

environment provides this possibility, but the main problem is related with the possibility of creating parallel applications within this environment as, until now, there was no implementation of standard (PVM, MPI) parallel programming libraries for Delphi.
MPI-Delphi constitutes, therefore, a first approach for Delphi's 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 user's interface creation.
MPI-Delphi interface is more suitable to some specific kind of problems, such as monitoring long execution time parallel programs or computationally intensive graphical simulations. Currently, we are working in the application of MPI-Delphi interface in several fields, such as medium and high level computer vision, memory and calculus intensive machine learning algorithms and several classical statistical problems [11]. Besides, MPI-Delphi has revealed as a good tool for research, because the development of new algorithms can be done quickly and, therefore, time inverted in the debugging of such algorithms is reduced. The P-EDR algorithm constitutes a nice example of this last affirmation.

References

1. M. Acacio, O. Cánovas, J. M. García and P. E. López-de-Teruel. *An Evaluation of Parallel Computing in PC Clusters with Fast Ethernet.* In Procc. of the ACPC 99, Salzburg (Austria), Feb. 1999.
2. J. C. Browne, J. Dongarra, S. Hyder, K. Moore and P. Newton. *Visual Programming and Parallel Computing.* Technical Report TR94-229, University of Texas, 1994.
3. M. Bubak, W. Funika and J. Moscinski. *Tuning the Performance of Parallel Programs on NOWs Using Performance Analysis Tool.* In Procc. of Applied Parallel Computing, LNCS, vol. 1541, pp. 43-47, Springer-Verlag, 1998.
4. W. Gropp, E. Lusk and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* The MIT Press, 1994.
5. B. Lester. *The Art of Parallel Programming.* Englewood Cliffs, New Jersey: Prentice Hall. 1993.
6. P.E. López-de-Teruel, J.M. García, M. Acacio and O. Cánovas. *P-EDR: An Algorithm for Parallel Implementation of Parzen Density Estimation from Uncertain Observations.* In Procc. of the 2nd Merged IPPS/SPDP, Puerto Rico, April 1999.
7. L.B. Lucy. *An Iterative Technique for the Rectification of the Observed Distributions.* The Astronomical Journal. Vol. 79, No. 6, pages 745-754, 1974.
8. U. Maier and G. Stellner. *Distributed Resource Management for Parallel Applications in Networks of Workstations.* In HPCN Europe 1997, LNCS vol. 1225, pp. 462-471, Springer-Verlag, 1997.
9. J. Meireles Marinho. *WMPI Home Page.* http://dsg.dei.uc.pt/~fafe/w32mpi/.
10. J. Piernas, A. Flores and J. M. García. *Analyzing the Performance of MPI in a Cluster of Workstations Based on Fast Ethernet.* In 4th Euro PVM/MPI Meeting, LNCS, vol. 1332, pp. 17-24, Springer-Verlag, 1997.
11. I. Pitas. *Parallel Algorithms for Digital Image Processing, Computer Vision and Neuronal Networks.* John Wiley & Sons, 1993.
12. A. Ruiz, P. E. López-de-Teruel y M. C. Garrido. *Kernel Density Estimation from Heterogeneous Indirect Observation.* Learning 98, Madrid, Sept. 1998.
13. 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 4th Euro PVM/MPI Meeting, LNCS, vol. 1332, pp. 49-53, Springer-Verlag, 1997.
14. R. Wismüller, T. Ludwig, A. Bode, R. Borgeest, S. Lamberts, M. Oberhuber, C. Röder and G. Stellner. *The Tool-set Project: Towards an Integrated Tool Environment for Parallel Programming.* In Procc. of 2nd Sino-German Workshop APPT'97, pp. 9-16, Germany, Sept. 1997.