A Debugger Interface for Parallel Programs

Denise Stringhini[1 3], Philippe O. A. Navaux[1], Jacques Chassin de Kergommeaux[2]
[1] Instituto de Informática
UFRGS - Universidade Federal do Rio Grande do Sul
Porto Alegre/RS, Brazil
{string, navaux@inf.ufrgs.br}
[2] Laboratoire Informatique et Distribution Montbonnot Saint Martin, France
{Jacques.Chassin-de-Kergommeaux@imag.fr}
[3] Faculdade de Informática
Universidade Luterana do Brasil (ULBRA)
Gravataí/RS, Brazil

Abstract—

This paper describes the debugger interface that has been developed to provide a complete debugging tool for parallel application programmers. PADI (Parallel Debugger Interface) is a symbolic on-line debugging interface whose main goal is to provide easy interaction and intuitive interface for programmers. To achieve these goals, PADI implements visualization support and selection mechanism. The visualization support helps users following the changes of process states during the debugging session as well as it gives access to all processes that are running under the control of the debugging environment. The selection mechanism helps users choosing the processes they want to control and visualize.

Keywords— **parallel debugging, on-line debugging, visualization, parallel programming, p/t sets**

I. Introduction

Parallel programming is undoubtedly more complex than serial programming. The control of multiple processes and their interactions are the main reasons for such complexity. Despite there are some existing tools that address the development phase of parallel programs, the complexity is often passed on parallel tools, that is, the tools are not very easy to use. Thus there is a need for actually easy-to-use environments and tools for parallel programming. In spite of the existence of interesting tools, including a number of commercial ones, their use remains insufficient, partly because of the complexity of utilizing some of them, partly because some of them are constructed for specific platforms. Therefore, there remains a lot of room for improvements of existing tools or development of more supportive ones.

Probably, the most required parallel tool by programmers is a parallel debugger. In turn, parallel debugging tools are among the most complex to develop and this explains perhaps why few of such tools have been commonly used so far. This paper describes a contribution to the field of parallel debugging. It concerns a debugger interface to parallel programs - PADI, whose main goal is to provide an intuitive and easy-to-use interface.

Parallel programs can incur basically into four types of errors: traditional serial ones, bad algorithms, deadlocks and race conditions. According to [BUH96], the most frequent ones are the serial errors, followed by the others in the presented order. At a first moment, PADI is concerned in help to find these most frequent types of erros in parallel programs. Meanwhile, considering that dealocks and race conditions are the errors most difficult to find, in the future PADI will also offer some mechanisms to help the detection of these kind of errors.

PADI is an interface for a parallel on-line symbolic debugger. On-line debugging is concerned with providing access to program symbols, like variables and registers. On-line debugging differs from off-line debugging by the interaction mode with the program execution. On-line debugging has a direct interaction with the application while off-line debugging interacts with a trace file recorded during the original execution of the parallel application. On-line debugging is similar to traditional serial debugging and, consequently, easier to use for the majority of programmers. However it does not address the non-determinism of parallel executions. Examples of on-line parallel debuggers are TotalView [ETN01] and DETOP [WIS96].

Event-based debuggers mainly address this non-determinism by permitting deterministic replay of non-deterministic programs [MCD89, LEB87], thus allowing cyclic debugging of such programs. On-line and event-based debuggers are complementary: on-line debuggers can be used alone for deterministic programs or during deterministic replay of non-deterministic programs [LEU92]. In the sequel of this article we will not be concerned with the non-determinism of some parallel executions, supposing that we are either debugging deterministic programs or that PADI can be used in conjunction with an event-based execution replay tool. Examples of such tools are MAD [KRA97] and Pajé [CHA00] (that helps in performance analysis).

Adding a visualization element to a parallel on-line debugger is a key to make it easy-to-use. Visualization is mainly useful to show how many and which processes are under the control of the debugger and even to show the changes in their execution status. Visualization can also be used to access processes and their contents.

Combined with visualization, a selection mechanism can add some flexibility to debugging tools. Basically, it consists in maintaining some processes/threads sets and directing parallel debugging commands to the selected set. PADI implements these mechanisms in order to provide flexibility to the interface, since it allows users to specify the targets of each distributed debugging command.

This paper describes the user interaction with the tool as well as some key concepts and implementation details. First, related work on parallel debugging is described. The section that follows the related work presents the PADI tool and its main design issues. After that, the basics of PADI architecture are outlined. Before the conclusion, the user interaction with PADI prototype is explained, in order to show PADI's main characteristics.

II. Related work

In spite of being (at least apparently) not active any more, the High Performance Debugging Forum (HPDF) [HPD00] contributed to parallel debugging with several important definitions. Created to define a standard for parallel debugging, HPDF covered many areas, although nothing concerned specifically with a Graphical User Interface (GUI) was defined.

One of the most important HPDF concepts concerned with parallelism is the *processes/threads set* (or p/t set) concept. According to HPDF, it provides the foundation for extending the semantics of serial debugger operations to a form suitable for parallel debuggers, allowing a debugger command to be applied to a whole collection of processes/threads, rather than to just one process or thread at a time. A **target p/t set** can be defined by selecting one, many or all processes/threads from the application. Considering a console debugger (without a GUI), this set can be defined by adding the processes/threads labels to the parallel debugger command. A default set can also be defined, that is the **current p/t set** (used when the programmer does not give any p/t label(s)). There is yet another type of p/t set, that is the **affected p/t set**, which defines the p/t set that will actually be affected by a command (it depends on the validity of the operation). Other useful definitions concerning control program execution and others are in the draft [HPD00].

One of the most widely used parallel debuggers is TotalView [ETN01]. It is a parallel symbolic on-line debugger offering many useful functionalities, such as a *root window* containing all the processes being debugged during a session and a *process window*, allowing processes inspection (the root window provides access to process windows). With respect to the p/t sets defined by HPDF, TotalView is less flexible since it does not allow the definition of subsets of processes, although it implements the *flexible breakpoints*, shared by parent processes and their children. Although it uses colors to highlight some important information during the debugging process, TotalView does not provide any kind of graphic visualization to express what is happening.

Another interesting tool is the p2d2 debugger [HOO99]. Its interface consists of a single window allowing programmers to coordinate up to 256 processes. It satisfies the two main criteria addressed here which are the implementation of p/t sets and visualization. It has a *process grid* where all processes being debugged are located in specific positions as icons that represents their status as defined by the user. The only drawback of this approach is the difficulty of locating or identifying a specific process in a grid containing a lot of them (in spite of a mechanism called *focus column*, that highlights the selected column and displays information about it in a specific area). With respect to p/t sets, p2d2 implements the *control set* mechanism that allows the definition of group of processes that will be controlled by the debugger. This group is highlighted in the grid, but the grid continues to show all processes.

Node Prism [SIS94] for Connection Machine (CM5) offers a language for defining named node sets. Besides specification by lists and ranges of nodes, set membership may be based on expressions involving program data. Using any legal source-language expression on each node, the node becomes a member of the set if the result is true. There is a window that maintains all defined sets and the user may choose one of them to be the current one. Regarding visualization, it provides filtered textual messages about processes states and actions and a *where tree*. The tree is a generalization of a back trace for multiprocessors. It groups together traces of processors that make calls to the same functions, from the same point in the SPMD code. It also provides data visualizers, that are graphical representations for arrays and array-valued expressions.

Finally, we introduce Fiddle [CUN98], that is the infrastructure level of PADI. Fiddle (previously named PDBG) is a framework that supports parallel and distributed debugging. It provides a debugging environment where clients (PADI, in this case) can make calls to perform any kind of parallel debugging command. This is achieved by coordinating process level debuggers and providing client communication with them. Fiddle is an Application Program Interface (API) and as such does not include any visualization, although it provides a console, useful for tests. Fiddle leaves the implementation of p/t sets to the client level.

Although there exists some useful debugging tools like the ones named here, there still exists the need for really intuitive and easy-to-use parallel debuggers. The design of PADI was made to excel simplicity for user interaction instead of having a great concern with scalability (that seems to be the main goal of existing parallel debugging tools, e.g. Node Prism and p2d2). Even that the scalability of such tools is an impor-

tant feature (in fact, it is in some level addressed by PADI), the begginers in the *art* of parallel programming normally work with few processors/processes and their main difficulty is to understand what is happening with them. Besides, often a program that works well with few processes will work fine with many more, thus, in cases like this, easy-to-use characteristic is more convenient then scalability. PADI intend to address intuitiveness by some design choices explained in this paper.

III. Design and implementation of PADI

The design choices of PADI that make it easy-to-use can be summarized as follow:

- the debugging environment is similar to the existing sequential ones, making it more familiar to the users;
- the basic debugging commands are accessible directly by buttons on the interface (small use of pop-up menus and use of icons/tooltips instead);
- the process visualization represents processes as easy to identify units from where users can access their contents;
- only selected processes are visualized, which cleans up the visualization area and turns debugging more intuitive since only visible processes will receive distributed commands;
- PADI makes clear distinction between the two main levels of hierarchy of a debugging tool:
  - one level for the distributed features, represented by the main window
  - and the other for the individual processes features, represented by processes' windows instances, one for each desired process.

Parallel on-line debugging tools include mainly two levels: coordination and process level (e.g., Fiddle/PDBG [CUN98]). The coordination level gives an overview of the application being debugged as well as an access to the process level. The process level in turn, provides access to the code, variables, stack, etc., of each process. In spite of the extra coordination level, this structure of parallel on-line debuggers is similar to what is provided by sequential debuggers. The coordination level is specific to parallel debuggers and therefore more interesting to explore while developing a parallel debugger. PADI implements these two levels.

The main goal of PADI is to explore the coordination level in a parallel debugger interface, making it intuitive and easy-to-use. In order to achieve this goal, PADI has its coordination level implemented as a main window having three main functions: distributed debugging commands interface, process visualization and process selection. These features are combined to provide means to select processes, to visualize only the selected ones and to distribute required commands only to these selected processes.

The process level is similar to what is provided by sequential debuggers. In PADI, this level provides almost the same commands as the coordination level. The difference lies in the target processes. While the targets of the coordination level are the selected processes, in the process level, only the owner of the window will receive a debugging command. In other words, commands from the process level in PADI have priority over the ones originating from the coordination level. This was done in order to make the tool more flexible, since the user is not forced to always use the selection mechanism as well as he can work individually with opened Process Views.

The programming models accepted by PADI are even SPMD (Single Program Multiple Data) or MPMD (Multiple Program Multiple Data). Threads are also allowed, even that at the moment they have no special treatment. PADI is being developed to work first with parallel programs writen in MPI [PAC97], PVM [GEI94] and DECK [BAR00].

IV. User interaction with PADI

The main idea in PADI is to separate, at least at the interface level, the distributed actions from the sequential ones. This was done by making a clear distinction between the two debugging levels explained before: coordination and process levels. Thus, PADI provides two main interface views: the Main View and the Process View. The first one is responsible for all actions that are distributed while the second one is responsible for all sequential symbolic inspections.

Since parallelism is the main *focus* of PADI and it is represented by the coordination level, we will *focus* in this section on the user interaction with the Main View. Figure 1 shows the Main View interface of the PADI prototype. The distributed debugging commands are available directly from the interface as well as the group selection mechanism. The **Processes Area** is where the parallel application processes are presented. The currently available distributed commands are (they are in the **Commands** bar that is below the **Menu** bar in figure 1):

- *load*
- *kill*
- *attach*
- *detach*
- *run*
- *step*
- *next*
- *continue*
- *finish*
- *set breakpoint*
- *delete breakpoint*
- *display variable*
- *undisplay variable*
- *set variable*

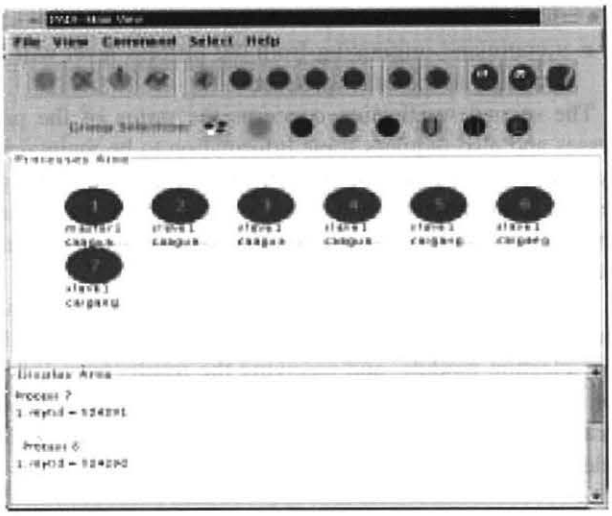

Fig. 1. PADI's Main View (coordination level).

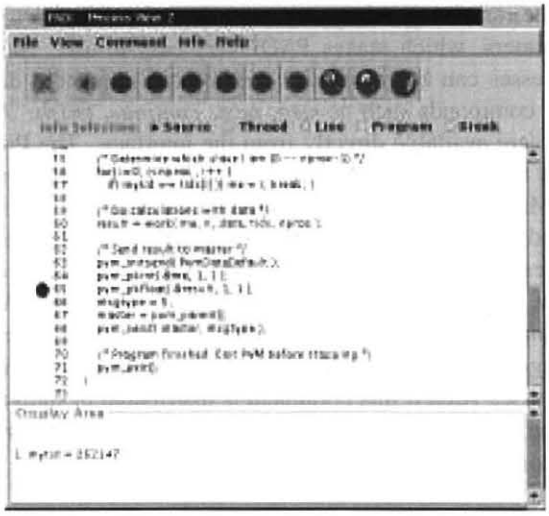

Fig. 2. Process View (process level).

Figure 1 shows a snapshot of an execution of a farmer/workers program written in PVM [GEI94]. A farmer process was loaded and executed until it created six worker processes, that were attached to PADI. The numbers inside the icons are their ids for the PADI environment.

The attachment of processes can be done in two ways: process by process or by executable name. In process by process mode, the user will be presented to a list of processes (available by a *ps* command) where it is possible to select the one that will be attached. On the other way, it is possible to provide an executable name that will be used to do a multiple attach: all processes with that name will be automatically attached. Another information that is necessary is the host where the process or processes are executing. PADI uses a host file (*padi_hosts*) where users can provide a list of available host names.

In order to debug a parallel application, the first thing to do is to **load** its main executable and/or to attach its processes. Once the parallel application is loaded, it is *ready* to be executed. In order to be able to debug it in step-by-step mode, it is necessary to **set a breakpoint** and to **run** it until this breakpoint. The new processes created by the application can be **attached** to the debugging environment so that they can also be debugged in step-by-step mode. Once new processes are attached, they also appear in the **Processes area**. Attached processes start normally in *stopped* state. The stopped state is the one that allows any debugging command to be executed by any process.

At any time during the debugging session, it is possible to select a group, using the **Group Selection** bar (third bar in figure 1). The pre-defined groups present in this bar are currently related to the process states or defined by the user:

- *all*
- *ready (green)*
- *running (blue)*
- *stopped (red)*
- *terminated (black)*
- *user*
- *not user*
- *exec*

The first one selects all processes holded by PADI. The following four groups (*ready, running, stopped, terminated*) are based on the processes states when the group was selected. The *user selection* is a little bit different. When a user selects a process in the **Processes area**, it is possible to select or unselect the process. Processes selected by the user receive a black border, indicating that they are part of the *user group*. When the user selects the User button on the **Group Selection** bar (the green "U" ellipse), the marked processes will be currently selected. On the other hand, if the Not-user button is selected (the black "U" ellipse), the processes not marked will be currently selected. Even if there exists more sophisticated mechanisms like the ones based on attributes (implemented in Node Prism and p2d2), this one simplifies the use of PADI, since it is possible to establish a dual debugging operation that characterizes a great number of parallel applications (like the farmer/workers type). Finally, the *exec* group contains all processes that have a given executable name.

Once a user selects one of these selection criteria, only the selected processes will appear in the **Processes area** and will receive valid debugging commands. At any time, multiple Process Views (figure 2) can be opened by selecting any process from the Main View's Process Area.

The remaining debugging commands are familiar to programmers, which makes PADI very easy-to-use. Stopped processes can be resumed by traditional sequential debugging commands such as *step, next, continue, finish*. All of them are available directly from the interface. The Process View has non-distributed versions of the same commands as the Main View (except *load*, *attach* and *detach*). If they are called from Main View, they are applied to all the selected processes. Otherwise, if they are called from the Process View, they are applied only to the caller process. When a command is about to be applied, the selected processes change their color to gray, so it is possible to visualize at Processes Area it this command is distributed or individual. At the same time, the processes maintain the green color until the command is actually performed by PADI.

V. The architecture of PADI

As mentioned before, PADI uses Fiddle as a low level debugging engine. Fiddle controls the distributed processes (attached to process level debuggers) and provides client communication with them by sending debugging commands and receiving/interpreting results. PADI is not just a GUI for Fiddle since it provides functionalities such as distributed commands, visualization and process selection.

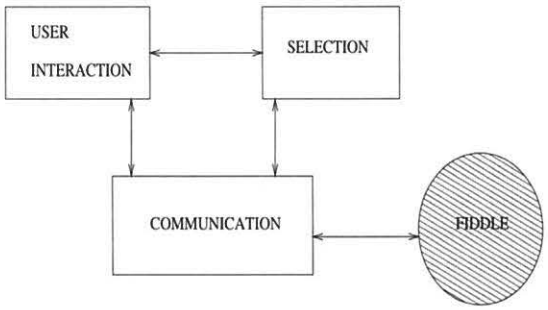

Fig. 3. Main blocks of the PADI structure.

Figure 3 presents the basic modules of PADI. The *interaction* module captures the user requests and displays their results after completion by the system.

The *selection* module controls the selection mechanism by performing some verifications before sending a request (debugging command) to the next module. The first verification is based on selected processes and is related to HPDF current p/t set: it defines which processes will (probably) receive the debugging command. Thus, this logical module is responsible for maintaining information about the processes present in the current p/t set. There are two alternatives in order to redirect a command to the processes:

- a command request originating from the coordination level will be sent only to the selected processes, where this selection is the one done by the user;
- a command originating from the process level will be sent only to the caller process.

The second verification concerns the status of the processes and also requires some information to be maintained by PADI. This verification is related to the HPDF affected p/t set: the command will be sent only to the processes that are able to perform the command. This prevents invalid commands to be sent to the next levels.

The *communication* module is the client of Fiddle. It interacts with Fiddle by sending requests, already filtered by the selection module, and receiving the results from Fiddle for each sent request. The results are then passed to the interaction module to be displayed to the user as well as to the selection module, if some status change took place.

A. Implementation details

The implementation uses the Java language [HOR99], which supports nicely the PADI architecture. Processes are represented by objects that encapsulate all the information needed to allow the verifications described above. These objects are instances from the *Proc()* class, that also has methods that are invocated when an event generated by a debugging process takes place. These methods are responsible to show such events into the interface.

The *Proc()* class contains the status information verified before a command is sent to Fiddle. In addition to this class, there is a set of classes that implement the actions needed by the commands: the *XxxAction()* classes, where "Xxx" is the name of the command. Basically, there is one class for each debugging command. These classes implement the status/selection verification (consulting the *Proc()* class) and mount a message that corresponds to the command. This message is then sent to an object that implements the communication with Fiddle (communication block in figure 3).

Another important component of the system is the processes' server (the *ProcsServer()* class), that is an object that maintains references to all the objects representing the processes being debugged. When a debugging event occurs, PADI initially receives, together with the information about the event, the Fiddle's tid of the process that caused the event. Then, the server is required to send the object's reference of the process based on that tid. After that, the appropriate method to the event detected is invocated at the *Proc()* object that represents this process in PADI context. Also, its state is adjusted. For example, if the process has finished its execution, its state is changed to *terminated*.

B. Processing flow in PADI

Figure 4 exemplifies the processing flow in PADI. The objects and methods were simplified, the goal is just to describe the main algorithm. The example shows a *run* command in-

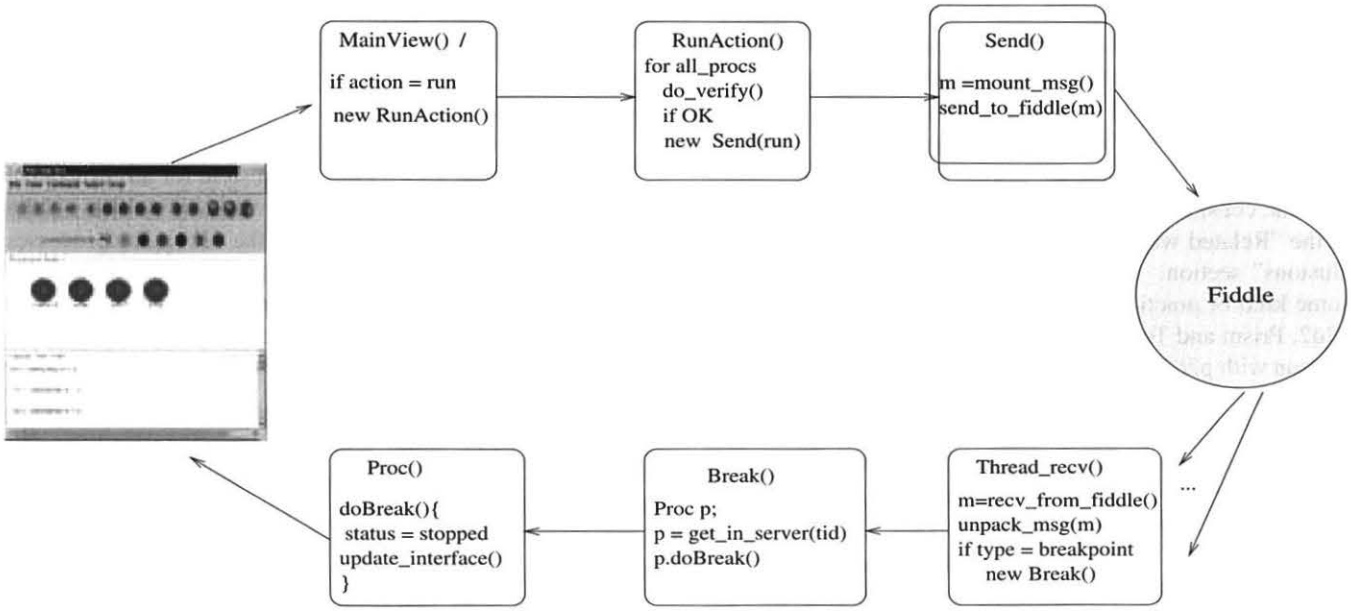

Fig. 4: Example of the processing flow in PADI: the boxes are the objects and the arrows are the relationship between them (creation or method invocation). The superior row of objects represents the processing flow to send a command (*run* in this case) to Fiddle. The inferior row represents the processing flow that occurs when an event from Fiddle (a breakpoint hiting in this case) is received until it is showed in PADI.

vocated by the interface and its processing flow until it is sent to Fiddle. The objects responsible for the interface control (*MainView()* or *ProcView()*) receive the command and create the corresponding *action* object (the *RunAction()*, in this case) to perform the verifications (status and selection). For each process that passes the verification, a *Send()* object will be created that mounts a message with required information about the command (the tid of the process and the run arguments in this case) and sends it to Fiddle.

After the sending of the run command an asynchronous event can occur, like hitting a breakpoint. This situation and the event back flow is also depicted in figure 4. There is a thread (named in figure 4 as the *Thread_recv()* object) that receives events like the arrival of a process at a breakpoint and creates the appropriate object to treat the event (the *Break()* object in the example). Then, this class gets the object reference to the process that caused the event and invokes the corresponding method in the *Proc()* class. This method (*doBreak()* in the example) updates the status of the process (that is *stopped* when arriving in a breakpoint) and also updates the interface: change the color of the process in the **Processes Area** and mark the corresponding line at source code (if the **Process View** is opened and showing the source code).

At this point it is important to note that commands like run can be distributed, what will create a number of *Send()* objects, one for each process. On the other hand, the responses are centralized in one thread that will receive all the asynchronous events from Fiddle. This centralized thread is used in order to establish a communication with Fiddle via a *pipe* mechanism. Meanwhile, Fiddle is being modified to accept Java clients, so the pipe is a temporary solution.

The PADI processes' objects (instances of *Proc()*) are created by *load* or *attach* commands. Then, when a debugging command, except of these two, is started in the interface, processes' objects states are verified. If the command was originated in a process view, that saves the tid of the owner, only the process' object holding this tid is verified and the command is sent only to it. If the command was originated from the main view, all processes' objects, whose references are maintained by the server, are verified and commands will be sent only to the tids that satisfy the conditions described earlier. This way of proceeding may seem heavy, but the advantage is that it is made locally. In other words, it avoids that a command that cannot be executed by a process in any way be sent through the network and be renegated by the target process. For example, a process whose state is *terminated* cannot be stepped. A *step* command will be intercepted by PADI locally, avoiding unnecessary network traffic.

VI. Preliminarly results

The PADI prototype can already be used such that it is possible to formulate some parctical comparisons with some existing tools. In the present phase of the work, the most important characteristics to be analyzed from a practical point of

view are the interface intuitiveness, the programming models accepted and the platform. The goal of PADI is to be an easy to use tool available to popular parallel programming libraries (like PVM and MPI) and platforms (Linux clusters). Also, it will be available without costs and will be used for further research in on-line debugging and monitoring areas.

Some considerations about other tools were already done at the "Related work" section and are retaken in the "Conclusions" section. This one is more concerned in making some kind of practical comparisons with knowed tools like p2d2, Prism and TotalView. Unfortunately, a practical comparison with p2d2 is not possible, since it is not available for download and its use is restrict to NASA (see the p2d2 site in [HOO99]). Prism was formely conceived at Thinking Machine Corporation and now is part of Sun HPC environment [SUN01], hence it is not available for Linux clusters. Both accept PVM and MPI programming models.

Totalview is a commercial tool, but it has an evaluation distribution for Linux clusters and others (available in [ETN01]). As a wide used commercial tool, TotalView is full implemented and has a lot of useful features. A comparison, at least for while, can be made only at interface level, that is the primary goal of PADI.

Let's consider the same PVM farmer/workers example presented at section "User interaction with PADI". When debugging that example with PADI it is easy to separate the farmer (master1 in figure 1) and the workers (slaves) into two groups. This can be done by selecting the farmer as *user* (just selecting its ellipse with the mouse) and choosing the *User group* to select it as the current group or choosing the *Not user* to select the workers.

The notion of a group in Totalview is less flexible. There are only two types of groups: the *control group* and the *share group*. The control group includes the parent process and all related processes. The share group is the set of processes within a control group that share the same source code. When debugging the same example with the Totalview defaults it is possible to experiment the effect of a control group by stepping the program. If the *Step group* option from the pop-up menu is choosed, all processes will do the step. Thus, the step can only be done individually or by all processes in the control group (that are all processes - farmer and workers - in this case).

The default breakpoint semantics is different in TotalView. When setting a breakpoint in a process it will be shared among processes with the same source code. By default, breakpoints follow the shared group semantics. The manual explains how to change the properties of an action point, but (at least apparently) this feature was not available in the tested evaluation version of TotalView (Linux x86 TotalView 4.1.0-1).

One of the main design choices of PADI is to separate the interface into two different kinds of views: one for parallel commands and other for individual commands. Such a structure clarifies the semantics of the debugger. In PADI it is possible to visualize all processes that will be affected by a debugging command before it is actually sent to execution. The PADI's Main View is the responsible for all distributed commands. The counterpart of this view in TotalView is the *Root window*, that is a textual list of the debugging processes and its states. However, all debugging commands are avalable by the pop-up menu from the processes views.

VII. Conclusion

PADI is a parallel on-line debugger interface whose main goal is to provide intuitive parallel debugging facilities. In order to achieve this goal, the two levels of a parallel debugger (coordination and process levels) were defined and implemented separately. Thus, the process level was made completely familiar to programmers, since it is very similar to traditional sequential debuggers. In addition, the coordination level of PADI was developed to embody parallel features, like distributed commands, selection mechanism and parallel visualization.

The coordination level of PADI is represented by its Main View. Several design choices contribute to making PADI intuitive and easy-to-use. The most important is the fact that the Main View embodies all the parallel commands. Other choices contributing to the goal of intuitiveness and simplicity include the fact that parallel commands, similar to traditional, sequential ones, were made directly available as buttons. Another interesting feature is the definition of groups of processes, that allow programmers to select processes to receive parallel commands. This is very useful since applications can have many processes performing different tasks during execution. Finally, visualization of parallel processes makes the tool very easy-to-use, since users can easily identify processes and access their process level view (the Process View) through their icons in the visualization area (Processes Area of the Main View).

Comparing PADI with tools already mentioned in this paper, shows that existing tools inspired some important design decisions (even to adapt some good ideas or to choose a different design).

The first one is the decision of separating, at user level (interface), the distributed commands from the serial ones. This approach is different from TotalView's approach since it is not possible to send any distributed debugging command from TotalView's Main Window, but just to choose the processes to be debugged. PADI's approach proved to be clearer and more practical, since one window (the Main Window) concentrates all the distributed actions, including the distributed debugging commands.

One of these distributed actions is the possibility of select-

ing processes. This feature, part of HPD standard, proved to be useful to debug parallel programs. It allows debugging groups of processes with similar characteristics as well as selecting processes suspected to be erroneous by the programmer. This feature was also implemented successfully in the p2d2 and Node Prism parallel debuggers. The main difference here is the simplicity of PADI, since selection can be done directly from buttons in the Main View.

In conjunction with the selection mechanism is the visualization of processes. Visualization has been applied in different areas of computer science, mainly to make interfaces more user-friendly. p2d2 created a very smart visualization grid, but it does not seem to scale well when lots of processes are being debugged (the identification of processes becomes difficult when the number of processes grows). The PADI approach uses the selection mechanism as a filter to processes visualization, clearing the Processes Area and showing just processes being inspected. Besides, the PADI visualization area includes information like process states (by color), process names or pids and tids (internal ids).

Besides improving the prototype in order to prepare a version for initial distribution, future work includes providing different types of visualizations (currently there exists only one) and to develop runtime system interaction, so that PADI can receive specific information from a specific system and show this information in the interface (e.g., process creation, send and receive events, etc.).

REFERENCES

[BAR00] M.BARRETO, R. ÁVILA AND P. NAVAUX "The MultiCluster Model to the Integrated Use of Multiple Workstation Clusters". In *Proceedings of the 3rd Workshop on Personal Computer Based Networks of Workstations* (PCNOW 2000), Cancun, volume 1800 of Lecture Notes in Computer Science, pages 71-800, Amsterdam, The Netherlands, 2000. Springer-Verlag.

[BUH96] P. A. BUHR ET AL, "KDB: a multi-threaded debugger for multi-threaded applications". In *Symposium on Parallel and Distributed Tools*, pages 80-87, Philadelphia, USA, 1996.

[CHA00] J. CHASSIN DE KERGOMMEAUX, B. DE OLIVEIRA STEIN, P. BERNARD, "Pajé, an interactive visualization tool for tuning multi-threaded parallel applications", *Parallel Computing 26*, 10, aug 2000, p. 1253–1274.

[CUN98] J.C. CUNHA, J. LOURENCO, J. VIEIRA,B. MOSCAO, AND D. PEREIRA. "A Framework to Support Parallel and Distributed Debugging". In *Proceedings of the International Conference on High-Performance Computing and Networking (HPCN'98)*, volume 1401 of Lecture Notes on Computer Science, pages 708-717, Amsterdam, The Netherlands, April 1998. Springer-Verlag.

[ETN01] Etnus Inc. TotalView Debugger, available by www in http://www.etnus.com/products/totalview (Apr. 2001)

[GEI94] A. GEIST ET AL. "PVM: Parallel and Virtual Machine- A User's Guide and Tutorial for Networked Parallel Computing". London: MIT, 1994

[HOR99] C. S. HORSTMANN AND G. CORNELL "Core Java 2". Volumes I and II, Sun Microsystems Press, 1999

[HPD00] High Performance Debugging Forum, HPD Version 1 Standard: Command Interface for Parallel Debuggers, Sept. 1998. Available by www in http://www.ptools.org/hpdf/draft (in sept. 2000)

[HOO99] R. HOOD. "The p2d2 Project: Building a Portable Distributed Debugger". *Proceedings of SPDT 96: SIGMETRICS Symposium on Parallel and Distributed Tools*. ACM Inc., 1996 (p2d2 site available by www in: http://www.nas.nasa.gov/Groups/Tools/p2d2/ - dec. 1999)

[KRA97] D. KRANZLMÜLLER, S. GRABNER AND J. VOLKERT. "Debugging with the MAD Environment". *Parallel Computing*, v. 23, p. 199-217, Feb. 1997

[LEB87] T. LEBLANC, J. MELLOR-CRUMMEY, "Debugging Parallel Programs with Instant Replay", *IEEE Transactions on Computers C-36*, 4, 1987, p. 471–481.

[LEU92] E. LEU, A. SCHIPER, "Execution replay: a mechanism for integrating a visualization tool with a symbolic debugger", *in : CONPAR 92 - VAPP V*, Y. Robert, L. Bougé, M. Cosnard, D. Trystram (ed.), *LNCS, 634*, Springer-Verlag, september 1992.

[MCD89] C. E. MCDOWELL, D. P. HELMBOLD, "Debugging Concurrent Programs", *ACM Computing Surveys 21*, 4, December 1989, p. 593–622.

[PAC97] P. PACHECO "Parallel Programming with MPI", Morgan Kaufmann Inc, 1997.

[SIS94] S. SISTARE, D. ALLEN, R. BOWKER, K. JOURDENAIS, J. SIMONS AND R. TITLE. "A Scalable Debugger for Massively Parallel Message-Passing Programs" *IEEE Parallel and Distributed Technology*, v. 2, n. 2, p. 50-56, 1994

[SUN01] Sun HPC ClusterTools 3.1 TM Available at http://www.sun.com/software/hpc/overview.html (july 2001)

[WIS96] R. WISMUELLER, M. OBERHUBER, J. KRAMMER AND O. HANSEN. "Interactive debugging and performance analysis of massively parallel applications", *Parallel Computing*, v. 22(3), pp. 415-442, March 1996