A Graphical Development and Debugging Environment for Parallel Programs

Péter Kacsuk*, José C. Cunha**
Gábor Dózsa*, João Lourenço** Tibor Fadgyas*, Tiago Antão**

*KFKI-MSZKI Research Institute
for Measurement and Computing Techniques
of the Hungarian Academy of Sciences
P.O.Box 49, H-1525 Budapest, Hungary
{kacsuk, dozsa, fadgyas}@sunserv.kfki.hu

**Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática
2825 Monte Caparica, Portugal
{jcc, jml, tra}@di.fct.unl.pt

Abstract

To provide high-level graphical support for PVM (Parallel Virtual Machine) based program development, a complex programming environment (GRADE) is being developed. GRADE currently provides tools to construct, execute, debug, monitor and visualise message-passing parallel programs. It offers high-level graphical programming abstraction mechanism to construct parallel applications by introducing a new graphical language called GRAPNEL. GRADE also provides the programmer with the same graphical user interface during the program design and debugging stages. A distributed debugging engine (DDBG) assists the user in debugging GRAPNEL programs on distributed memory computer architectures. Tape/PVM and PROVE support the performance monitoring and visualization of parallel programs developed in the GRADE environment.

1 Introduction

As local area computer networks have become a basic part of todays computing infrastructure, more and more people encounter the possibility to exploit the available computational power of heterogeneous networks of computers. The most widely used paradigm for implementing applications on such distributed systems is the message passing (MP) concept, and it is expected to be

the most common approach for the next few years. The MP paradigm has two fundamental advantages: simplicity and efficiency. The concept is quite simple to understand and can be implemented efficiently on different distributed systems.

A number of MP interfaces are available today, but one of the most popular is the PVM (Parallel Virtual Machine [17]) software package. PVM permits a user to configure his own virtual computer by hooking together a heterogeneous collection of UNIX based machines, on which the user has a valid login and are accessible over some network. The user views PVM as a loosely coupled distributed memory computer programmed in C or FORTRAN with message-passing extensions.

Although the concept of the MP paradigm is quite simple, the development of parallel programs is much more difficult than that of sequential ones, because of the extra tasks arising due to the communication and synchronisation of processes. The PVM system provides a low-level interface that enables to write and execute parallel applications but misses high-level support which could make this work acceptable easy and efficient. In the framework of two Copernicus projects (SEPP [33] and HPCTI), a complex programming environment (GRADE) is being developed to assist the whole cycle of parallel program development based on the PVM system.

GRADE stands for Graphical Application Development Environment and currently consists of the following tools as main components:

- GRED: A graphical editor to write parallel applications. The editor supports the syntax of the graphical language GRAPNEL [19] [13].
- GRP2C: A precompiler to produce the C code with PVM function calls of the graphical program.
- DDBG: A distributed debugger [8].
- Tape/PVM: A monitoring tool to generate trace file during execution of the PVM application [25][26] (developed independently at LMC-IMAG, Grenoble, France).
- PROVE: A visualisation tool to analyse and interpret Tape/PVM the trace file information and present them to the programmer graphically.

The scheme of the program development cycle in GRADE is depicted in Figure 1. Rectangles represent the different tools (GRED, DDBG, etc.). Ovals denote data (files) used by the system. Furthermore, object libraries (GRAPNEL, Tape/PVM, etc.) are represented as rounded boxes. As a first step the user applies the GRED graphical editor to design and construct the parallel program written in a special graphical programming language called GRAPNEL. The GRED editor creates the so-called GRP file from the GRAPNEL program. The GRP file contains all the information necessary to restore the

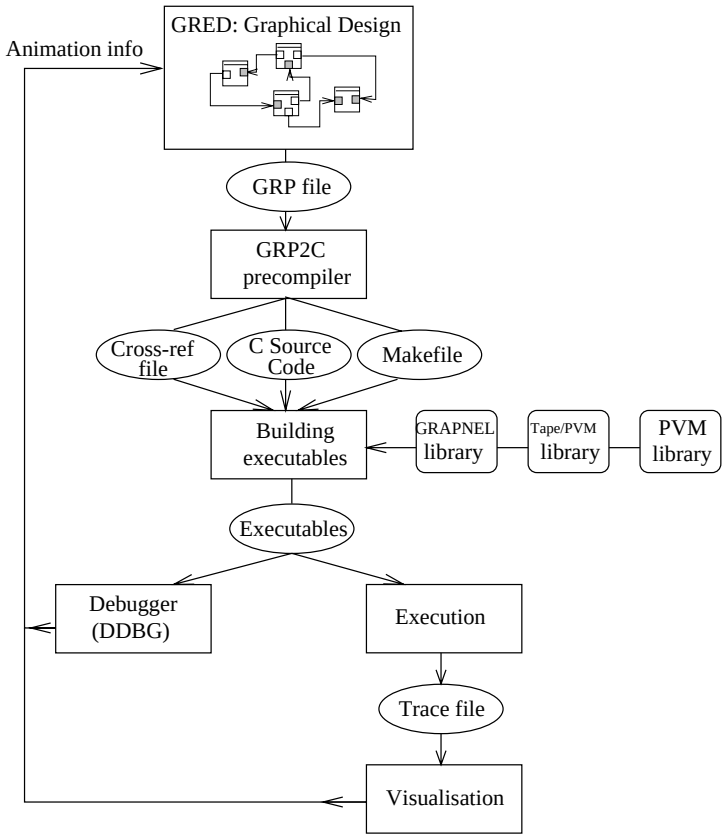

Fig. 1. Development Cycle in GRADE

program graph for further editing and to compile the GRAPNEL program into a C+PVM code. The latter is the task of the GR2PC precompiler which additionally creates the necessary makefile and a special cross-reference file to support the graphical debugging and animation of the program. The executable code is generated by the builder that applies three libraries:

- GRAPNEL library: to hide the details of PVM at the GRAPNEL level;
- PVM library: to realize interprocess and interprocessor communications;
- Tape/PVM library: to instrument PVM calls for run time monitoring and event trace collection.

The executable code is loaded to the processors and is executed either in debugging mode or trace mode. In debugging mode, the DDBG distributed debugger controls the execution of the program by providing commands to create breakpoints, step-by-step execution, animation, etc. In trace mode, a trace file is generated containing all the trace events defined by the user. These events are visualized by the PROVE graphical visualization tool assisting the user in spotting performance bottlenecks in the GRAPNEL programs.

The role of libraries mentioned above may require some further explanations. Figure 2 illustrates that they are organized in a hierarchy where each library

function is called from a higher level library or from the generated code. At the bottom the PVM library is used to establish interconnection between different processes. This library is covered by the GRAPNEL library which has the advantage that in case of porting GRADE to the MPI system [18], the generated code could remain the same and only the GRAPNEL library should be modified according to the requirements of the MPI library. If monitoring is not needed (like in case of Process B in Figure 2) the GRAPNEL library functions directly call the PVM library. However, if monitoring is necessary, the Tape/PVM library is inserted between the GRAPNEL and PVM libraries for many function calls of the GRAPNEL library in order to generate the event trace file.

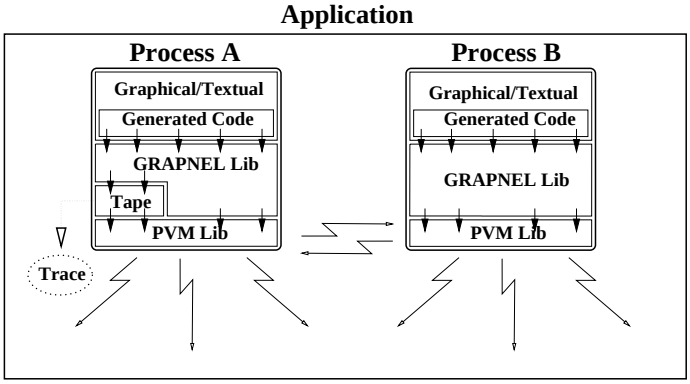

Fig. 2. Library Calls in GRAPNEL Processes

The library functions actually represent different message layers in a GRADE application. On top there is the GRAPNEL level where only high level communication actions (*send* and *receive*) are visible. The GRAPNEL libraries transforms this message layer to the system level where the necessary low level system services are provided to realise the high level communication actions. Such low level system services include the appearance of a service process responsible for spawning the processes and administrating their task identifiers (TIDs). The high level communication actions are realised as bidirectional message pairs at the system level. If monitoring is needed the Tape/PVM level provides extra communication facilities to collect the local traces. Similarly, in case of debugging daemon processes are created to handle messages necessary for controlling the debugged execution mode. Finally, at the lowest PVM level all necessary messages appear including system, monitor, debugging, etc. messages. The four message layers of the GRADE applications are depicted in Figure 3.

In the current paper we describe these tools and their relationships. One of the main advantage of the GRADE approach stems from the combination of the graphical design tool and the graphical debugging/animation environment. The user can debug and animate the program in the same graphical environment that was used to design it. Accordingly, one of the main highlights of

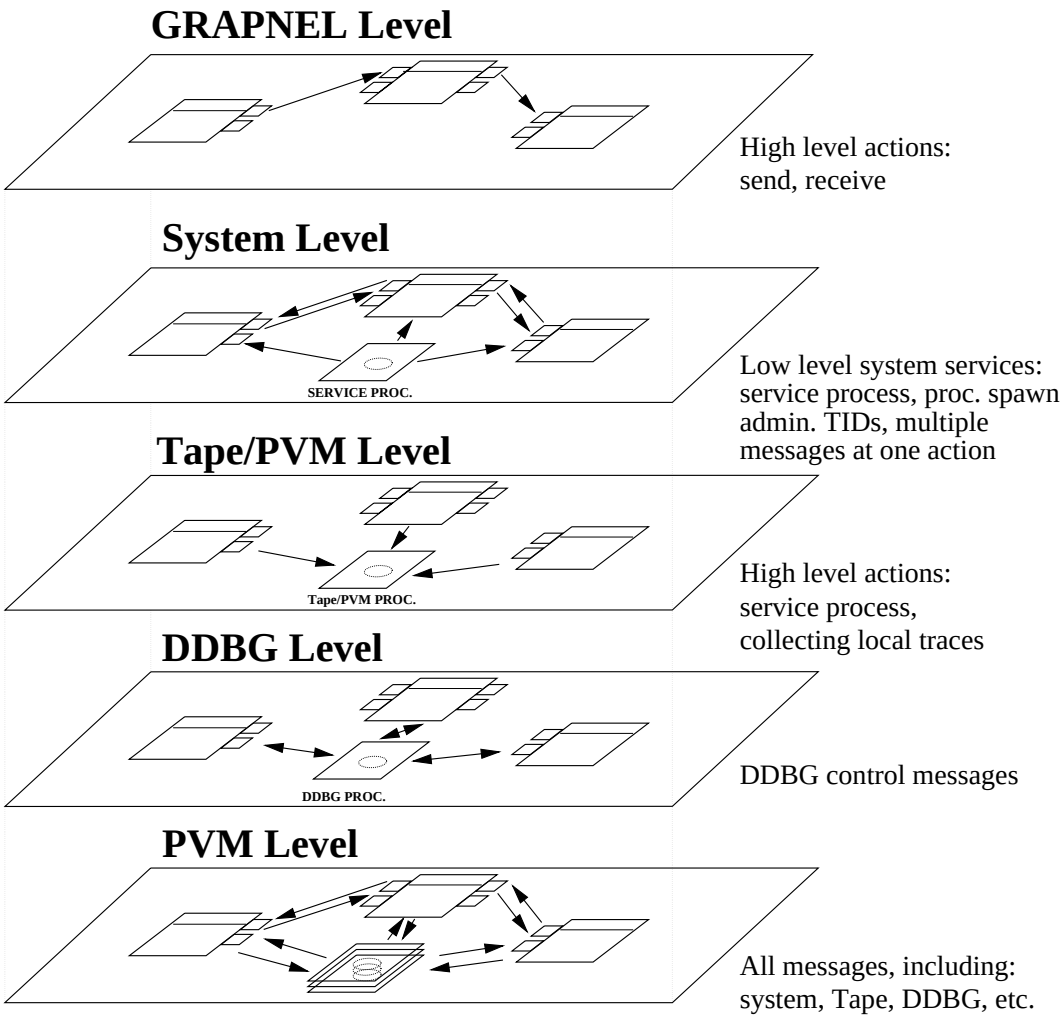

Fig. 3. Message Layers of GRADE Applications

the paper is the explanation of the combined use of the distributed debugger and the graphical programming environment. Similar work has been described with a flavour of simulation modeling of parallel systems in [10] and [11].

The structure of the paper: Section 2 describes the GRAPNEL graphical programming language and its graphical editor called GRED. Section 3 summarizes how the GRAPNEL programs are compiled. Section 4 gives detailed explanation on how the distributed debugger DDBG is used for debugging and animation of GRAPNEL programs. Section 5 introduces the main features of the Tape/PVM monitoring system. Finally, Section 6 shows the PROVE performance visualization tool.

2 The GRED Graphical Editor

In GRADE, parallel programs can be constructed according to the syntax and semantics of GRAPNEL language by using the GRED editor. GRED is built on the top of the X-Window system. The code is written in C++ and the Interviews 3.1 library is applied to program the X interface.

As the first step in the development cycle, the programmer must write the code of the parallel application. We state that at this phase graphics are needed to provide high-level support and abstraction mechanism. In the design and implementation of parallel programs the user is often encouraged to specify computational issues by drawing graphs and to take these drawings into consideration when it is necessary to track the run-time behaviour of the application. A good example of such a computational issue can be the process communication graph. It's no use making these drawings by paper and pencil when it is possible to draw them by using a comfortable and intelligent graphical program editor. However, there are program parts that nobody wants to draw, e.g. sequential functions or procedures that have nothing to do with message-passing or multiprocessing at all. It is only the outline of the parallel code which is worth being described by graphics; the low level details are more concise and more comfortable to be defined in ordinary textual way. That is the main idea behind the GRAPNEL language supported by the GRED editor. Processes, communication operations and communication connections must be described graphically, while low-level code of the processes can be written in C (or in FORTRAN in the future). A graphical outline of the application can facilitate the work of the programmer in the design phase as well as in the debugging and performance tuning phases. In the design phase, structured code can be written—or better to say, can be drawn—above the level of individual processes (i.e. above the level of C 'main' functions by means of PVM) by grouping processes which can be viewed as one unit into one node, or by using predefined communication topology templates. During the debugging or performance tuning phase, the programmer has the global view of message-passing related part of the application, where the errors or the bottlenecks can be located much faster then having and observing only the textual code of several processes using ordinary text editors and debuggers.

Due to the lack of space, we just summarize some basic idea of the GRAPNEL programming model in the following instead of giving the exact definitions of the language elements. The interested reader should refer to [19].

2.1 The Programming Model

The GRAPNEL programming model is entirely based on the message-passing paradigm. The programmer can define processes performing computations independently and interacting only by sending and receiving messages. A process can be either a single unit, or a member of a process group. A process group is an ordered collection of processes. Both processes and process groups are defined graphically as boxes. Process groups can be used in two important ways. Firstly, they can be used to specify the scope of a collective communication operation, such as multicast (i.e. a message can be received by all members of a group). Secondly, they can be used as an abstraction mechanism to support structured design at the level of processes, i.e. processes can be put into a group to be managed together as one unit. Since process groups can be nested, they support hierarchical design. Communications among processes are either point-to-point or group communications, and can be blocking or non-blocking ones. Communication always takes place via communication ports which can belong either to processes or process groups, and which are connected by channels. To ensure that the form of the transmitted data matches at both the sender and receiver sides, each port has its own protocol. Inside the processes, input and output operations represent the two fundamental types of communication actions. They are represented graphically, and they are joined with the port symbols on which the communication operations are to take place. The user can define the data to be sent or received by simple listing the names of the program variables where the data should be stored or should come from.

In GRAPNEL programs, three hierarchical design levels are distinguished. At the top level the outline of the whole application is described graphically with respect to communication connections among the processes, while at lowest level the textual code fragments are given (remember, that GRAPNEL is a hybrid language). At the middle level the send and receive operations are defined inside the code of a process graphically. These levels are represented by different types of windows shown in Figure 4.

2.1.1 Application Level

This is the top level of the parallel application. Processes, process groups, communication ports and connections among the processes must be defined here graphically, but the functionalities (i.e. the code) of the processes are hidden at this level. In other words, processes are viewed as black boxes here and we are interested in only the messages transferred among them. The largest window in Figure 4 represents the Application Window of GRED which can be used to construct the graphical code at this level.

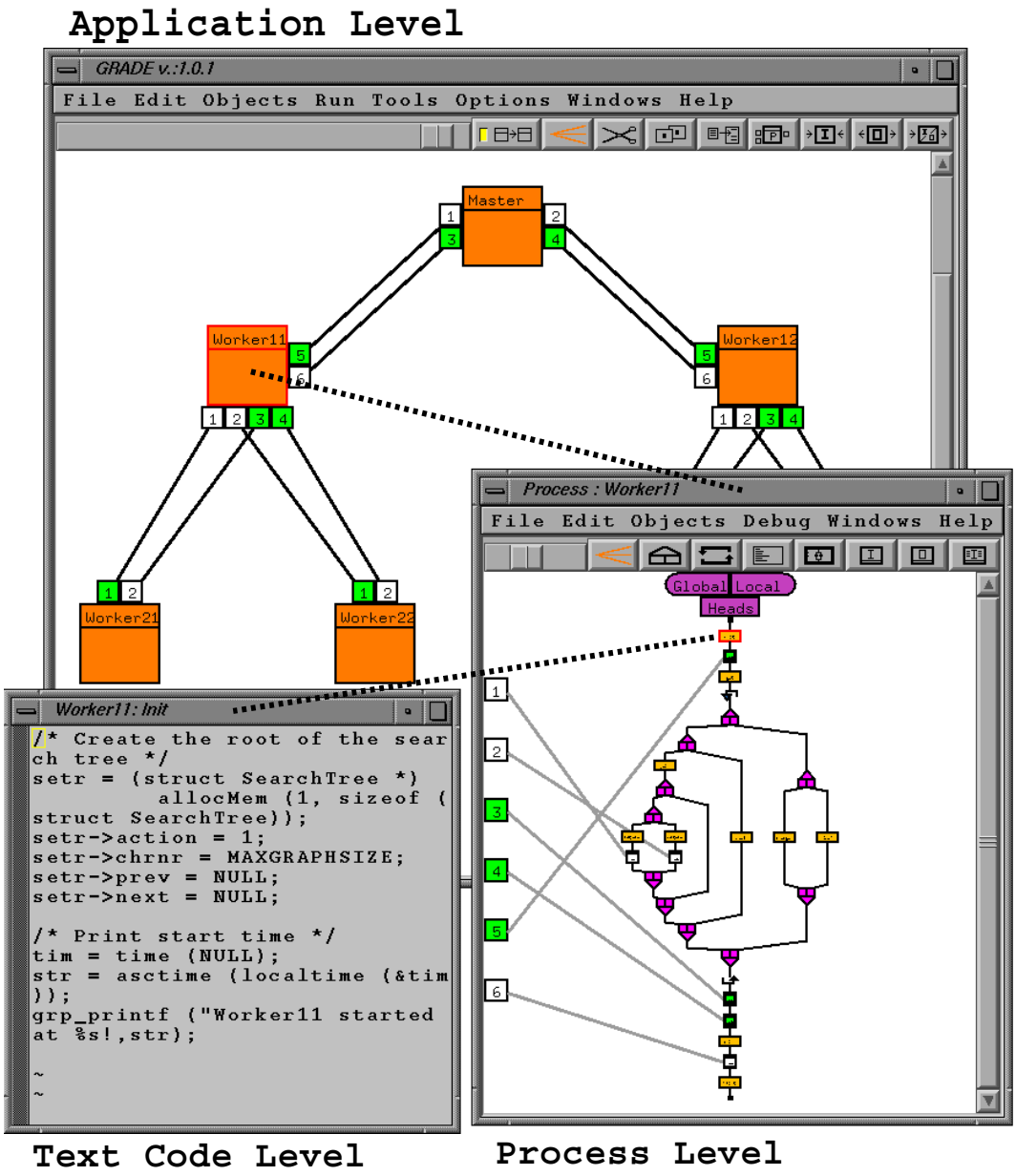

Fig. 4. GRAPNEL Design Levels

2.1.2 Process Level

This level describes the message-passing related parts of the control flow of a process graphically. The point is that send and receive operations must appear graphically in the GRAPNEL code. This approach has two relevant advantages. Firstly, the programmers do not have to know the syntax of the underlying message-passing library, i.e. they can just put an icon into the control flow of the process and simple list the variables where the data is to come from (send) or where it is to be stored (receive), instead of having to write all

the necessary PVM function calls [1] (e.g. different types of pack and unpack routines, buffer management, etc.). They are generated by the GRP2C precompiler automatically based on the graphical information of the GRAPNEL code. Secondly, in the debugging and testing phase all message transfers can be animated in the graphical windows that can significantly ease the identification of message-passing related bugs in the program. GRAPNEL defines graphical symbols (icons) to describe loop constructs, conditional constructs and send or receive operations graphically. However, only those segments of the control flow must appear graphically which contain send or receive actions. The textual block symbol is defined for denoting an arbitrary large and sophisticated textual code fragment that does not contain any send or receive action, so which can be defined textually at a lower level (i.e. text code level). In order to support the structured design the graphical block symbol has been introduced at the process internal level. A graphical block icon denotes a control flow segment that does contain communication operations, therefore the content of the graphical block is defined graphically [2]. In GRED, the so called Process Window can be used to create the graphical code at this level (see the Process:Worker11 window in Figure 4).

2.1.3 Textual Code Level

It is the lowest level of the GRAPNEL code where the programmer can define the textual C code fragments. In the text code belonging to a textual block icon the programmer may call C functions that are defined either in library or in normal C files, thus C code written earlier can be reused easily. The text editor that is invoked by GRED to edit these text parts of the program can be defined by the user through UNIX environment variables (i.e. without recompiling the system). As a result, the programmer may use his favorite text editors (e.g. emacs, vi, etc.) in GRED. An example for a textual window (Worker11:Init) is depicted in Figure 4.

3 Compiling the GRAPNEL Programs

The programmer can save the code of his application into the so called GRP file which is a text file and contains all the information necessary to restore the graphical program in the GRED editor, or to produce the executables of the application. The executables are generated in two steps. In the first step, the pure C code of the GRAPNEL program is generated by the GRP2C

[1] Actually, no PVM functions are written in the GRAPNEL code at all.
[2] The graphical block can be viewed as some kind of graphical subroutine.

precompiler. Afterwards, a standard C compiler is invoked to produce the executable binaries from the C code.

3.1 The GRP2C Precompiler

GRP2C has been written in C++ and it comprises two main parts: the parser and the code generator. The parser has been implemented as a separate library with its own header files since it is integrated into both the GRP2C precompiler and the GRED editor. The *lex* and *yacc* standard UNIX tools have been used to generate the C code of the lexical and syntactical analyser routines based on a BNF (Backus-Naur Form) like description of the GRP syntax. The input of the parser is the GRP file and 'output' is an internal data structure that is used either by the editor to restore the graphical program on the screen or by the code generator to produce the C code. The code generator of the GRP2C precompiler produces three types of files: C files, a Makefile and a cross-reference file. We give a brief description about each type as follows.

3.1.1 C Files

A separate C file is produced for each process of the GRAPNEL program. Basically, the correspondence between a graphical GRAPNEL process and the C file generated for that process is quite straightforward: each program item in the GRAPNEL code has its own piece of code in the C file. For instance, in the case of a textual block, the code generator simple copies the contents of the block to the appropriate position in the C file. (With respect to this principle, it does not seem to be too difficult to support FORTRAN beside C, as it simple means that the programmer can define FORTRAN code for the GRAPNEL program items and the code generator puts these code fragments into a FORTRAN file.) However, the situation is a little bit more difficult concerning the communication operations. To define a send or receive operation at the Process Level, the programmer just lists the affected variables that determine the data to be transferred and connects the graphical send or receive icon with the appropriate port symbol that gives the target of the communication. The code generator must translate that information into PVM function calls, i.e. it must generate 'pack' or 'unpack' function calls for the listed variables and must use process 'tid' values and 'message tags' in the PVM send or receive calls instead of GRAPNEL channel numbers. This problem was one of the main reasons why we decided to create an interface library called GRAPNEL library between the PVM library and the generated C code of the GRAPNEL processes. The code generator places GRAPNEL library calls into the C files instead of direct PVM function calls. GRAPNEL library calls fit the communication related program items of the GRAPNEL

code. All the PVM function calls, buffers, tids, etc. appear only inside the GRAPNEL library. This approach has also the advantage that we can support other message passing systems (e.g. MPI [18]) easily in the future, as only the internal details of the GRAPNEL library should be rewritten.

3.1.2 Makefile

Beside C files, the code generator creates a 'Makefile' for the GRAPNEL program. This Makefile can be used to produce the executable form of the application by invoking the UNIX make command. The GRP2C precompiler uses the GNU autoconf utility to gain the necessary information required to produce such Makefile (e.g. what kind of C compiler is available in a particular host machine). As GRAPNEL programs run in heterogeneous environments, the Makefiles on different hosts can differ significantly. Therefore, the GRP file is copied to each PVM host and GRP2C is invoked by the system on each host simultaneously.

3.1.3 Cross Reference File

The third type of files produced by the code generator is the cross reference file. This file contains information about the connections between graphical program items in the GRAPNEL code and textual code segments in the C files (i.e. which line in the C file corresponds to a particular graphical program item in the GRAPNEL code). The cross reference file is very important for the graphical animation of the application or for error fixing when the executables are created from the C files.

3.2 Creating the Executables

Having got the C files and the Makefile for the GRAPNEL program on each PVM host, the programmer can invoke – from the GRADE environment – the UNIX make utility that invokes the C compiler and the linker in turns on each affected host. However, the programmer can happen to make mistakes in the textual parts of the GRAPNEL code (e.g. inside a textual block), so he needs feedback about the compilation of the C code. For this purpose the GRADE environment provides a message window, which opens on the screen when the compilation process has started. This window collects and shows the error and warning messages from every host where the make command has been executed (there is a wrapper program for the make command that grabs the standard output of the make and sends it to the GRADE environment via an internet socket). In this way, the programmer can have information about all problems encountered by the C compilers and the linkers. When the user

chooses any of those messages with the pointing device, the GRED editor will highlight the graphical item to which that particular message belongs, furthermore, it will highlight the incorrect user defined textual code in the corresponding text window.

After building the executables, the programmer can start the PVM daemons on the hosts and execute the parallel program. During execution, the process icons are coloured according to the start-termination information sent by the application server to the editor (i.e. the actual state of a process is reflected by the colour of its icon). If the parallel application does not behave as it was expected to, the user can invoke the distributed debugger integrated into the GRADE environment.

4 Debugging and Animating GRAPNEL Programs

GRED provides a high level interface for on-line debugging of GRAPNEL programs. The programmer can debug the application in exactly the same environment where it was constructed. Moreover, in the design of GRAPNEL language we particularly focused on the debugging aspects (i.e. the graphical outline of the program is extremely useful to locate communication related errors). For this purpose GRADE uses the services provided by the DDBG (Distributed DeBuGger), which applies a process-level debugger to supervise tasks of PVM applications at its lowest level. DDBG defines a set of C functions that can be embedded into other systems in the same way as it has happened in case of GRADE [8].

4.1 Debugging Parallel and Distributed Systems

Traditional sequential debugging techniques offer the following typical functionalities: cyclic interactive debugging, memory dumps, tracing, and breakpoints [29]. However, these techniques cannot be directly applied in a parallel and distributed environment. This is due to the following facts: parallel and distributed programs exhibit non-deterministic and non-reproducible behavior; lack of global state makes it very difficult to manage global predicates on the system state; and there is an intrusion effect of the debugging system upon the observed program.

The most immediate approach to support debugging functionalities in a parallel and distributed environment is through the collection of multiple sequential debuggers, each attached to an application process. This may provide similar commands as available in conventional debuggers, possibly extended to deal

with parallelism and communication. However, this does not solve any of the above difficulties.

In the past 10 years, several proposals have been made to address these problems [21] [1] [6] [3] [12] [27] [2] [32]. We are particularly interested in an approach that models the debugger as an event-based system. This provides several interesting characteristics: it uses a previously recorded event trace, in order to analyze the execution history, and to guide program replay with reproducible behavior, and so it can make use of (suitably adapted) conventional debugging techniques; it may rely upon monitoring techniques, for event generation and recording; it can benefit from optimizations that allow to reduce the amount of collected information, namely based on the instant replay technique [21] and so it can greatly reduce the intrusion effect; it eases the management involved in the global coordination of parallel processes and inspection of global system states.

Providing Basic Debugging Support to other Tools

A large diversity of debugging tools have been developed for distinct parallel and distributed programming languages, as well as for distinct parallel computer systems. In particular, the appearance of shared-memory and distributed-memory multiprocessors during the 80s has originated the need to develop specific debugging support, both at the level of the operating system and at the level of the communication libraries [1] [3] [2]. The problem with these debugging tools is that usually they are specific to a particular runtime or hardware environment, and as such they are very difficult to adapt to other parallel platforms. On the other hand, the evolution of parallel and distributed systems towards more user-friendly environments requires a very flexible software development platform for the experimentation with new programming models and the corresponding development tools, e.g. for monitoring, debugging, animation, visualization, and performance analysis. Several years of experimentation with these parallel computing platforms still show a need to provide a more unified framework to support the implementation of high-level debugging functionalities. This framework must address two fundamental issues:

- A well-defined interface must be provided to be used by high-level tools of the parallel development environment, namely graphical editors, graphical interfaces, runtime support systems for distinct parallel and distributed language models, and testing and high-level debugging tools;
- A well-defined interface must be provided to the underlying operating system and hardware platform, assuring portability and adaptability of the debugging support architecture, while still allowing efficient implementa-

tion on top of each specific physical environment.

In order to experiment with the issues involved in the design of a flexible and general-purpose debugging architecture, we have implemented an interface library of debugging primitives on top of the PVM [17] system. Besides providing the commands that are typically supported by conventional debuggers for sequential computation models, this interface provides the basic primitives to inspect and control distributed processes. From the user point of view, any application or tool can be linked with the interface library and access all the distributed debugging functionalities. From the implementation point of view, the current design has a distributed organization consisting of multiple monitor/debugger instances which are scattered on the nodes of a PVM platform.

One of the distinctive goals of our approach when designing and implementing the DDBG system was to provide a platform supporting easy experimentation with tool integration as far as debugging is concerned. Two main experiments were performed concerning the interfacing of DDBG with other parallel software development tools which exhibit very distinct functionalities. Besides the integration with the GRED tool that is described in this paper, the DDBG system was successfully composed with the STEPS testing tool [8]. One important issue of that experiment concerns reaching a close integration of static analysis and dynamic analysis methods in order to guarantee the final quality of the parallel and distributed software. Besides formal methods to assure the quality of parallel programs, systematic testing approaches play a very important role in this process. The development of a methodology and tool to aid the user in the process of identifying the paths which should be generated and tested, is a key component of an advanced testing and debugging environment. Although a detailed discussion is beyond the scope of this paper, an important aspect of this approach is to allow the testing and evaluation stages to be performed through a close interaction with the dynamic debugging tool. After the STEPS tool has identified interesting program paths, the DDBG tool is invoked in order to support user controlled execution of the paths under test, allowing the user to inspect program behavior at the desired level of abstraction and with the guarantee of the reproducibility of its execution [33] [20] [8].

4.2 The DDBG Architecture

In this section we present the interface provided by the distributed debugging tool called DDBG and its logical architecture. Its debugging functionalities may be summarized as follows:

– Dynamic attachment and deattachment of debugger instances to already

running distributed processes; control of remote debugger instances from a central debugging user interface;
- An interface library that gives access to such control of remote debuggers, and which can be used by high-level tools, like a graphical editor, and testing tools;
- An event trace is collected with minimal information to support program replay in PVM programs. This allows reproducible behavior and will make the debugging control commands available during a replay session; a checkpoint facility under replay mode will support execution replay from an intermediate point, instead of from the beginning of the program only.

Currently there is a working prototype implementing the first two of the above functionalities. Full support for program replay and checkpointing is under final development.

The DDBG is composed by a collection of process-level debuggers which are controlled by a distributed architecture that provides distributed debugging functionalities. This distributed architecture contains the following components: a main daemon, multiple local daemons, a debugging interface library, a text console and a graphical user interface. Any user tool can access the debugging engine as a client process that uses the debugging library to interact with the main debugging daemon. The main daemon manages all the interactions with the client process, by forwarding the debugging commands to the machines where the application processes are placed, and collecting their corresponding answers. This is achieved by having a local daemon on each machine that is responsible for the activation and control of multiple debugger instances located in that machine. Each application process can be dynamically attached (detached) to (from) a distinct debugger instance.

This architecture is illustrated in Figure 5 where its main components are shown (the *delayed answers* are explained in Section 4.3).

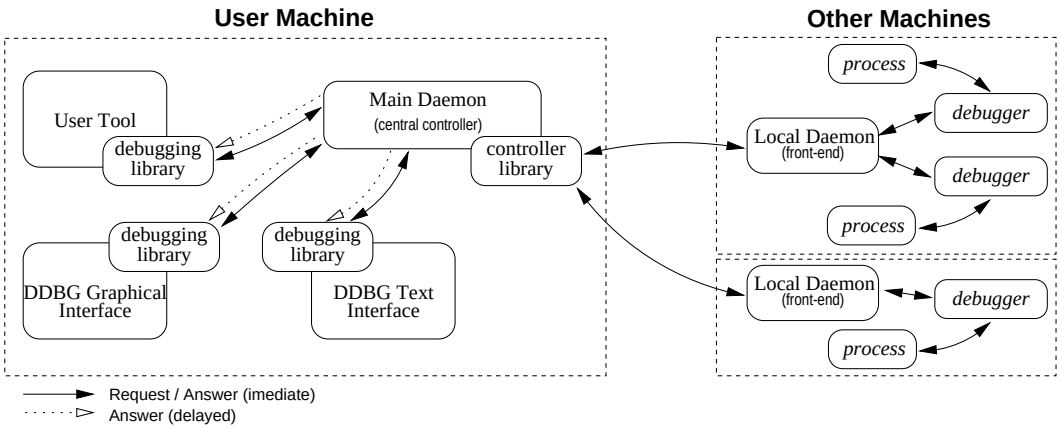

Fig. 5. The DDBG architecture

4.2.1 The Process-level Debugger

This component of the DDBG distributed debugger applies conventional debugging commands to each application process. It can be a proprietary debugger for a specific hardware/software architecture as well as a public domain one [3] .

Although for better exploitation of individual debugger capabilities user access should be provided to the particular features supported by each process-level debugger, this is in contradiction to the requirement of offering a coherent user view for the whole debugging environment. As a result of the experimentation with the integration of DDBG and other tools we have recently started to work towards a more generic distributed debugging command language than our current gdb-based version.

4.2.2 The Local Daemons

For the *main daemon*, each local daemon acts as a mirror of the process-level debuggers running in its node. For each debugger running in one specific node, the local daemons act as a mirror of the *client tools*.

The complexity of such a kind of component can vary, depending upon the level of abstraction provided by its interface protocol to the external clients (the *main daemon* in this case). If this protocol is low-level, as it really happens in the DDBG system, the complexity is very low as the process just has to forward the incoming commands from one external client to multiple debuggers running in that node, and vice-versa for the corresponding replies.

4.2.3 The Main Daemon

The *main daemon*, as the central component in the system, supports the following main activities:

- Start the local daemons in each node;
- Provide an interface to access the DDBG distributed debugger;
- Interpret and send the client (high-level) commands to the corresponding local daemon;
- Process all the replies given by the process-level debuggers (through the local daemons) and forward them to the client tool.

[3] Currently, we are using "*gdb*", the public domain debugger from the GNU Software Foundation.

During the initialization phase, the main daemon starts a local daemon in each machine of the computing environment. These local daemons will simplify the communication protocol between the process-level debuggers and the *main daemon*.

Multiple client tools can dynamically connect and disconnect to *the main daemon* (and to the debugging system). All these client tools share the same view of the debugging system (e.g. all the client tools know that breakpoint B is in process P, line number N) and all of them have access to the full system (e.g. any client tool can delete a breakpoint set by another tool). These characteristics provide a very high degree of flexibility in the use of the debugging system.

Depending on the level of abstraction that may be provided by the interface library (see section 4.3), the *main daemon* has more or less work for preprocessing the client commands and convert them to (a sequence of) debugger commands.

To support a distinct process-level debugger, a new parser is needed for processing its replies, involving a possible corresponding increase in the complexity of the *main daemon*. The need to accommodate these different needs may impose a limitation in the number of different debuggers supported by the current DDBG system. A new design for the DDBG architecture is currently under development in order to overcome this limitation.

4.3 The Debugging Interface Library

The detailed description of all library functions is beyond the scope of this paper but the most important functions are summarized in the following. (The interested reader can find the details of all library functions in [7].)

PVM uses *task ID's* (integers) to identify the processes, but a user application or tool may use specific *Process ID's* (strings) to do the same. In order to support the mapping between the user symbolic name and the PVM naming scheme, a name mapping function is provided. This allows any of the library primitives, as well as the corresponding user consoles, to refer to string or integer process identifiers, although in the following description we always use string identifiers.

Currently the communication between a client and the main daemon requires the client to invoke a *system call* **init()** [4] to initialize the library [5]. If there is

[4] Actually, all library function names are prefixed with dbg_.

[5] The current prototype assumes that the PVM system is already running.

no main debugging daemon running at the time the initialization is requested, it will be started automatically. This initialization also establishes a communication channel that will be used for future interactions between this client and the debugging engine [6] .

All the services provided by the debugging engine are classified as belonging to one of two classes:

- **Immediate answer services.** These services will either fail or are immediately executed by returning the relevant data as output parameter(s) to the corresponding *system call* function.
- **Delayed answer services.** These services —e.g. **next()**—will either fail with an immediate return, or they may take an unpredictable amount of time until its execution is finished. In the latter case, the corresponding library function also returns immediately to the calling process but informs the user about the unavailability of the data, which can later be collected by invoking the function **get_special_info()**, as defined below.

In principle the interaction with the debugging system can be completely transparent. The client just invokes library functions, and gets the corresponding returns in an immediate way or in a delayed way. In the latter case, the following function can be used:

- **int get_special_info(char *procid, struct code_info *info)**
  If there is returning data available (from a *delayed answer service*), the function will return a corresponding status indication with the user Process ID in `procid` and the data in `info`. Otherwise it will inform about the unavailability of the data.

This function has a non-blocking semantics so that the asynchronous execution of the user application that controls the debugging interface is allowed.

4.3.1 Managing breakpoints

Basic support is provided to control program execution through breakpoints which are currently only associated with individual processes. The function **set_break(char *procid, struct code_info *info)** sets a breakpoint on a given process, in the line/function that is specified in `info`. An unique `breakpoint id` is returned. A similar function sets a temporary breakpoint (one time only).

Breakpoints can be set conditionally, depending on the evaluation of an expres-

[6] Currently, a UNIX file descriptor is returned corresponding to an interprocess communication socket.

sion, by invoking **set_cond_break(char *procid, struct code_info *info, char *exp)**. If the expression `exp` evaluates to `TRUE`, a breakpoint is set in the specified process, in the line/function specified in `info` (conditional breakpoint). The expression is evaluated every time the breakpoint is reached. An unique `breakpoint id` is returned.

Watchpoints can also be specified for a given process by invoking the function **set_watch(char *procid, char *exp)**. This function sets a watchpoint on the given process `procid` such that the process will stop when the condition in `exp` will become `TRUE`.

Breakpoints can be temporarily disabled, enabled, cleared (permanently removed), or ignored a certain number of times by invoking corresponding library primitives.

4.3.2 Controlling the execution of the (debugged) processes

The library supports classical debugging commands to control the execution of each individual process in a detailed way. Using these commands, as well as the other commands that handle breakpoints, and display or update process information, it is possible to implement higher level debugging functionalities. This was used to implement the interfacing of DDBG to other tools [8].

The function **run(char *procid)** allows to start running a (previously spawned) process from the beginning, until a breakpoint is found or the expression of a watchpoint is true, or until the end, if none above the conditions occurs. The execution of a stopped process can be continued by invoking **continue(char *procid)**. The execution proceeds until one of the above mentioned conditions occurs. Another function **finish(char *procid)** allows to run a process until the currently selected stack frame returns, as defined by the `select_frame()` function. It is also possible to pop the selected stack frame without executing and return in `info` the relevant data to determine where the process was stopped.

Step by step execution is supported by the functions **next(char *procid)** and **step(char *procid)** which execute the code until the next instruction, by respectively executing subroutine code as one instruction only or as normal code.

Interrupting the execution of a process is supported by a call to the **interrupt(char *procid, struct code_info *info)**. The returned `info` contains information to determine where the process was stopped.

4.3.3 The Text Console

This component act as a client of the DDBG system, through a connection with the *main daemon* using the *debugging library*, and provides a command line interface and command parser to allow access to all the functionalities available through this *debugging library* (see figure 6).

As the system dynamically accepts connections and disconnections with multiple client tools, this text console can be started at any time and coexist with any other client tool in the DDBG system. This ability provides the user with different abstraction levels for debugging a program, each one supported by a specific client tool (see Section 4.4 for a description of an interface with high-level abstraction debugging concepts).

4.3.4 The Graphical User Interface

The aim for the graphical user interface for the DDBG system is to provide the user with an intuitive and easy-to-use interface to access the DDBG functionalities (see Figure 6).

Currently, there is a basic prototype which supports a directory browser of the processes under debugging, and for each one there is a *var watch* window. In this window, the user can specify some variable names, valid in the current execution context of the associated process. All out-of-context variables are automatically removed from the displayed list. The refreshment of the variable values is also done on explicit command, to avoid heavy communication between the *graphical UI* and the *main daemon*, for updating the variable values every time a process reaches a breakpoint.

4.4 The High-level Debugging Interface to the GRED editor

The GRADE environment is centered around the GRAPNEL model described in Section 2.1.

Concerning debugging, the GRED editor offers an user-friendly interface that allows to invoke debugging commands with reference to the graphical entities that are displayed in the user windows. On the other hand, there is a requirement to display in a convenient way the debugging outputs such that only GRAPNEL abstractions should be handled by the user at this level. This offers a very high-level interface to the user, such that the information on specific debugging commands is directly related to the GRAPNEL source program, e.g. by highlighting corresponding entities in the graphical representation, and their corresponding lines of source code in the textual program representation.

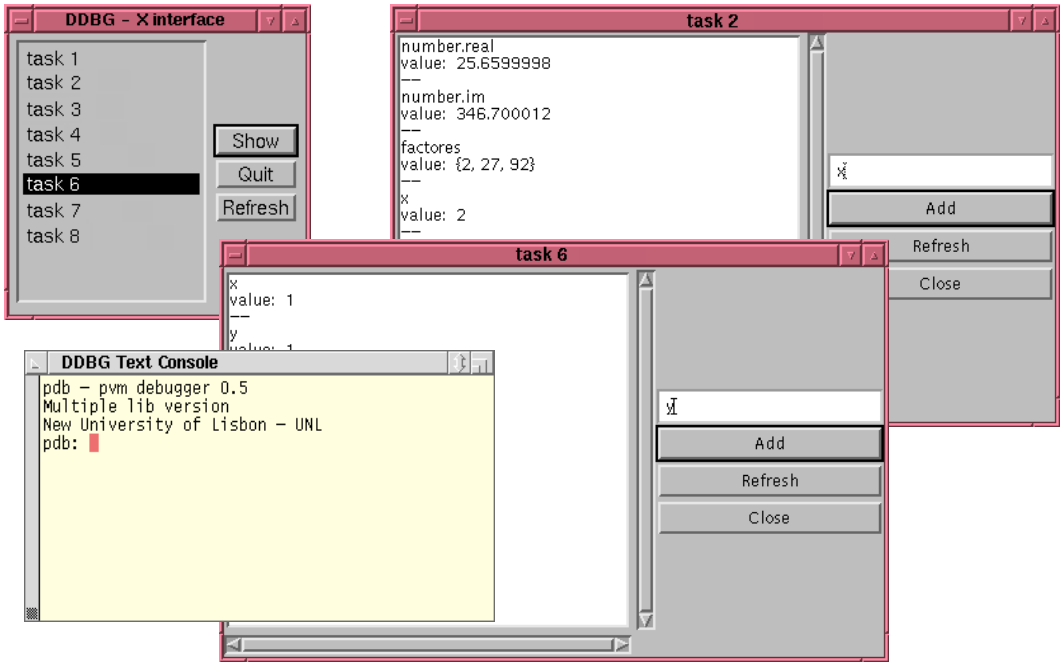

Fig. 6. The DDBG User Interface

During debugging, the programmer can use the usual way to control a process (e.g. 'run', 'step', 'continue', etc.) and to examine data at both graphical and textual level of the code. Breakpoints can belong to either graphical symbols at the process internal level or a specific line in the textual code of a graphical program item. A breakpoint on a graphical symbol is denoted as a filled circle at the top of that symbol, which circle is highlighted when the process actually stops at that point of the code. Furthermore, breakpoints can be set on a communication port symbol to stop the process every time it performs data transfer via that port. At process communication level every process icon is coloured dynamically according to the actual state of that process . When a breakpoint is reached by a process, then its icon is coloured at the process communication level to denote that the process is blocked. Similarly, the appropriate graphical box where the process has stopped at the process internal level is coloured in the same way. If that program item is a communication operation (i.e. send or receive) then the channel via which data has just been transferred is highlighted as well. Thus, the graphical level of the code can efficiently support to debug all message transfer among processes (i.e. all parallel activities in the application).

These high-level debugging functionalities are supported by interfacing the GRED editor with the DDBG system. The editor interface also accesses the DDBG graphical user interface which displays, under user control, the variables defined within the GRAPNEL structures. This X-based interface is automatically started when the user hits the debugger item under one of the menu options provided by the editor. Then a display is presented including

the user processes and the user can select the processes to be monitored, and individual variables within these processes as explained above.

The GRED editor runs as a process that handles asynchronously generated events, namely user interaction events. When interfacing the editor to the distributed debugger, this asynchronous mode of operation is handled by using the described function **get_special_info()**). A more efficient control is possible by having the editor directly accessing the interprocess communication socket that is created by the function **init()**. Although this is not so transparent when compared to the exclusive use of the above function **get_special_info()**, it allows the editor to selectively wait[7] on that socket.

5 Monitoring GRAPNEL Programs by TapePVM

Tape/PVM is a tool to generate event traces of PVM applications for post-mortem performance analysis [25]. Though Tape/PVM is integrated into the GRADE environment, it has been developed independently from the SEPP/HPCTI project, at LMC-IMAG, Grenoble, France.

Tape/PVM generates events at user application level by intercepting the original PVM library calls. A preprocessor is provided to instrument the user source code (C or FORTRAN) automatically, i.e. to exchange every PVM library call for the corresponding Tape/PVM call. Tape/PVM allows selective tracing; i.e. the user can specify before executing the instrumented code what kind of PVM calls he wants to be monitored . Moreover, the programmer can define additional events (i.e. user defined events) beside the PVM related ones.

6 Visualisation by PROVE

PROVE is a performance visualisation tool. It is an adaptation of the PACVIS [14] system for distributed memory computers[8] . PROVE is a post-mortem tool which reads the information from a tracefile that was generated during the application run. TAPE/PVM generates an event (i.e. a record in the trace file) whenever a PVM function call occurs. Additionally, the GRADE system also defines its own events related to the entering into sequential or communication blocks.

PROVE uses the X11 facilities and the OSF/Motif widget set for displaying

[7] Using the *select()* UNIX system call.
[8] PACVIS supports the shared memory model.

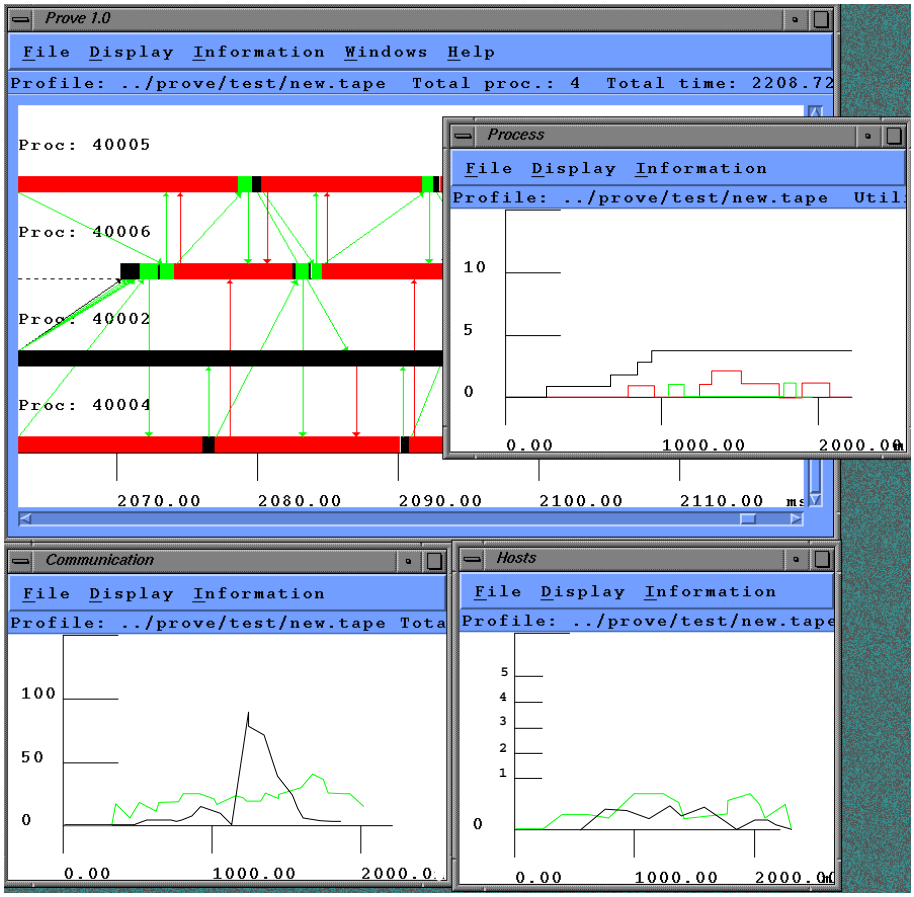

Fig. 7. A Sample PROVE Session

the required information. All required default values and constants (colors, default sizes, button names etc.) can be customized via X11 application resources. The most important features can be also selected by command line arguments. Fig. 7 depicts a sample PROVE session.

PROVE has several windows that display two-dimensional graphs (where the horizontal axis is always the time axis):

(i) **Behavior Window:** Displays the processes, their different states, interactions, start (spawn) and terminate events, communication events and ports, user marks, etc.
(ii) **Processor Window:** Displays the utilization of the processors (i.e. PVM hosts) during the runtime of the application program.
(iii) **Communication Window:** Displays the communication among the **processes**, especially the rates and the amount of data transferred between the processes.
(iv) **Hosts Window:** Displays the communication among the **hosts**, especially the rates and the amount of data transferred between the host computers (processors).

The **Behavior Window** is the central window of the PROVE tool and created on the startup of the program. The behavior graph is a two-dimensional representation of the program. The horizontal axis represents the program time while the executed processes are arranged along the vertical axis. Each process is represented by a horizontal line that shows the time during which the process was executed by a processor. The thick horizontal lines have several sections coloured differently showing the stages of the process (e.g. computation, communication, etc.). The number of processes and the total program runtime are displayed at the top of the window.

By default, only the process lines are displayed. However, some additional amount of information can be selected:

- **Start:** shows the new process creation. An arrow (black) starts from the parent process to the child process.
- **Stop:** shows when a process kills another process. A black arrow starts from the killer process to the victim process.
- **Wait:** the wait (blocked receive) dependencies are displayed by red arrows between the waiting process and the process waited for.
- **Send:** the data delivery is displayed by green arrows between the sender and receiver processes.
- **Ports:** as an additional information to the send/receive arrows the port numbers are also displayed. A port number appears inside a small rectangle attached to the horizontal bar of the sending/receiving process.
- **Mark:** the time marks created by the user are shown by blue vertical lines in the graph.

All these layers of information can be independently switched on and off and arbitrarily combined. The user can select/deselect specific communication events because they are assigned to different PVM message tags. Additionally, moving the mouse pointer within the graph lets the user zoom (i.e. enlarge) rectangular subsections of the graph.

The PROVE has also a simple animation feature. If it is working together with the GRED editor, it is able to highlight the icons on the GRED's windows. (See Fig. 8.) A single mouse click in the **Behavior Window** shows the user the relevant source code block in the GRED's **Process Window**. From the position of the cursor in the **Behavior Window** PROVE finds the actual process based on the vertical coordinate and the actual block in the process based on the time coordinate. PROVE uses the animating feature of GRED to highlight the icon belonging to the active block. By means of this feature the user can easily identify the relevant source code objects meanwhile he/she investigates the program behavior by PROVE.

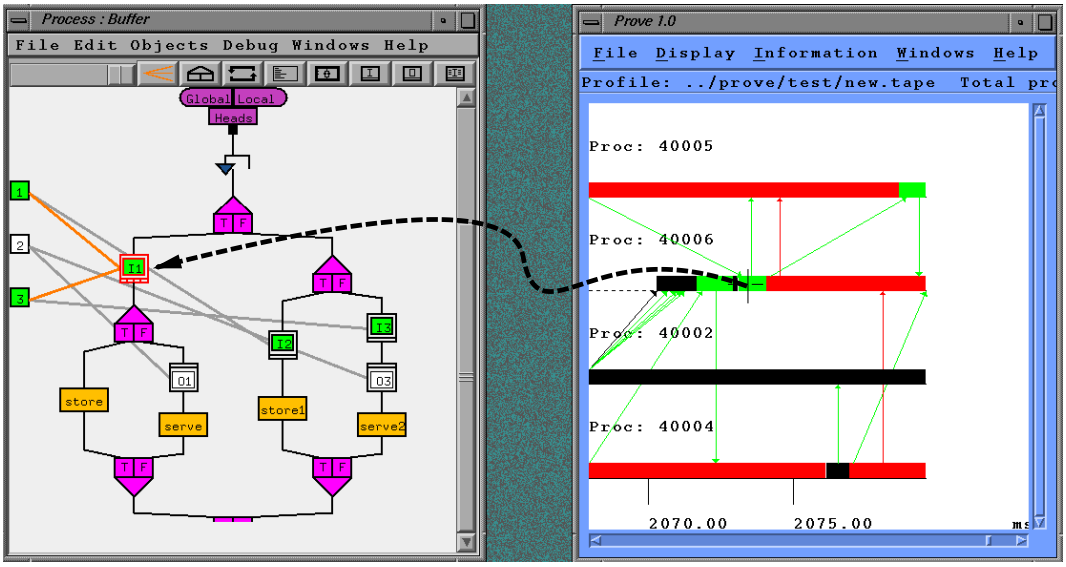

Fig. 8. Animation by GRED and PROVE

7 Related Works

A number of other visual parallel programming language and environment have been developed (e.g. HENCE [5], CODE [30], PSEE [23], Paralex [4], TRAPPER [31]). Most of them are based upon the idea that nodes represent computation, and arcs represent interactions (of some form) among nodes. The problem with the HENCE, CODE, Paralex and PSEE approaches is that they force computations to be split into separate nodes when communications occur or when branching decisions control communications (i.e. some kind of dataflow approach). This can result in complicated, awkward and large graphs. This problem does not arise in TRAPPER, which system is very close to GRADE concerning both purpose and functionality. However, the TRAPPER model is static as it has originally been designed for transputer systems and it does not offer an integrated debugger for on-line debugging and animating of the graphical parallel program.

In GRADE, all communication activities can be defined graphically without any restriction concerning their locations. Dynamic process creation and predefined topology templates are going to be implemented based on the definition of GRAPNEL language. Furthermore, the programmer can design and debug his/her parallel application by using the same graphical user interface and abstraction mechanism. Program tuning is supported by PROVE visualisation tool which cooperates with the same graphical user interface as well.

Conclusions

The more people encounter the possibility to exploit the available computational power of heterogeneous networks of computer, the more vital is the demand for high-level tools to assists the development of message-passing-based parallel applications. GRADE provides a complex programming environment where the user can develop a parallel program by using high level tools and abstractions without worrying about the low-level details of message-passing primitives. Structured program design is supported above the level of individual processes. All communication activities can be defined graphically without any restriction concerning their locations. A distributed debugger is fully integrated into the system, thus, the programmer can design and debug his/her parallel program by using the same graphical user interface. Program tuning is supported by PROVE visualisation tool which cooperates with the same graphical user interface as well.

The first evaluation of the GRADE environment revealed the modifications and extensions that are necessary in order to make it really comfortable and useful for parallel program development. We plan to implement a replay mechanism and to integrate a simulation and a systematic testing tool into the system.

Acknowledgment

This work is partly funded by the Commission of European Communities Contract Num: CIPA-C193-0251, CP-93-5383 and by the Hungarian National Committee for Technological Development (OMFB) in the framework of Austrian-Hungarian inter governmental cooperation under Project Num: B.52.

References

[1] *ACM Workshop on Parallel and Distributed Debugging.* ACM SIGPLAN Notices 24(1) (1988).

[2] *ACM/ONR Workshop on Parallel and Distributed Debugging.* ACM SIGPLAN Notices 28(12) (1993).

[3] *ACM/ONR Workshop on Parallel and Distributed Debugging.* ACM SIGPLAN Notices 26(12) (1991).

[4] O. Babaoglu and L. Alvisi and A. Amoroso and R. Davoli. Paralex: An Environment for Parallel Programming in Distributed Systems. in: *Proc. of ACM International Conference on Supercomputing* (1992)

[5] A, Beguelin, J.J. Dongarra, G.A. Geist, V.S. Sunderam. Visualization and Debugging in a Heterogeneous Environment. *IEEE Computer* 26(6)(1993).

[6] W. Cheung, J. Black, E. Manning. A framework for distributed debugging. *IEEE Software* 1 (1990).

[7] J.C. Cunha, J. Lourenço, T. Antão. The DDBG Distributed Debugger User's Guide. Technical Report, Universidade Nova de Lisboa, Portugal, 1996.

[8] J. C. Cunha, J. Lourenço, T. Antão. A Debugging Engine for Parallel and Distributed Environment. in: *Proc. of DAPSYS'96: 1st Austrian-Hungarian Workshop on Distributed and Parallel Systems* (Miskolc, Hungary, 1996) 111-118.

[9] J.C. Cunha, H. Krwaczyk, B. Wiszniewski, P. Mork, P. Kacsuk, E. Luque, L. Sutovska, L. Hluchy. Monitoring and Debugging Distributed Memory Systems. in: *Proc. of uP'94: The Eight Symposium on Microcomputer and Microprocessor Applications* (Budapest, Hungary, 1994).

[10] T. Delaitre, G.R. Justo, F. Spies, S. Winter. Simulation Modelling of Parallel Systems. in: *DAPSYS'96: 1st Austrian-Hungarian Workshop on Distributed and Parallel Systems* (Miskolc, Hungary, 1996) 25–32.

[11] T. Delaitre, F. Spies, S. Winter. Simulation Modelling of Parallel Systems in the EDPEPPS project. in: *Proceedings of the UKPAR'96 Conference* (1996).

[12] P.S. Dodd, C.V. Ravishankar. Monitoring and debugging distributed real-time programs. *Software—Practice and Experience* 22(10) (1992).

[13] G. Dózsa, T. Fadgyas, P. Kacsuk. GRAPNEL: A Graphical Programming Language for Parallel Programs. in: *μP'94: The Eighth Symposium on Microcomputer and Microprocessor Applications* (Budapest, 1994) 304–314.

[14] T. Fadgyas, W. Schreiner. Visualization of Parallel Programs: The PACVIS Visualization Tool. in: *Proc. of the 2nd Austrian-Hungarian Workshop on Transputer Applications* KFKI-1994-09/M,N Report (Budapest, 1994) 43-61.

[15] A. Fagot, J, Chassin-de-Kergommeaux. Optimized execution replay mechanism for RPC-based parallel programming models. Technical Report, IMAG, Jul. 1995.

[16] C. Fidge. Partial orders for parallel debugging. ACM Workshop on Parallel and Distributed Debugging, *ACM SIGPLAN Notices* 24(1) (1988).

[17] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing.* (MIT Press, 1994).

[18] W. Gropp, E. Lusk, A. Skjellum. *Using MPI : Portable Parallel Programming with the Message-Passing Interface* (MIT Press, 1994).

[19] P. Kacsuk, G. Dózsa, T. Fadgyas. Designing Parallel Programs by the Graphical Language GRAPNEL. *Microprocessing and Microprogramming* 41 (1996) 625-643.

[20] H. Krwaczyk, B. Wiszniewski Structural Testing of Parallel Software in STEPS in: *Proc. of the 1st SEIHPC Workshop, COPERNICUS Programme* (Braga, Portugal, 1996).

[21] T.J. LeBlanc, J-M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. on Computers* C-36(4) (1987).

[22] T. Ludwig, R. Wismuller, V. Sunderam, A. Bode. OMIS — on-line monitoring interface specification. LRR-TUM, Technical Univ. of Munich, Germany, and Emory Univ. USA, Feb 1996.

[23] E. Luque and R. Suppi and J. Sorribes. Overview and New Trend on PSEE. *IEEE Software* (1992).

[24] M. Mackey. *Program replay in PVM.* Hewlett-Packard, Concurrent Computing Department, H.P. Laboratories, May 1993.

[25] E. Maillet. Tape/PVM: An Efficient Performance Monitor for PVM applications. User Guide, at ftp://ftp.imag.fr/imag/APACHE/TAPE/

[26] E. Maillet. Issues in Performance Tracing with Tape/Pvm. in: *EuroPVM'95* (Lyon, 1995) 143–148.

[27] Y. Manabe, M. Imase. Global conditions in debugging distributed programs. *J. of Parallel and Distributed Computing* 15 (1992).

[28] D.C. Marinescu, J.E. Lumpp, Jr., T.L. Casavant, H.J. Spiegel. Models for monitoring and debugging tools for paralell and distributed software. *J. of Parallel and Distributed Computing* 9 (1990) 171–183.

[29] C.E. McDowell, D.P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys* 21(4) (1989).

[30] P. Newton and J.C. Browne The CODE 2.0 Graphical Parallel Programming Language. in: *Proc. of ACM International Conference on Supercomputing* (1992).

[31] C. Scheidler and L. Schafers TRAPPER: A Graphical Programming Environment for Industrial High-Performance Applications. in: *Proc. of PARLE'93: Parallel Architectures and Languages Europe* (Munich, Germany, 1993).

[32] J.J-P. Tsai, S.J.H. Yang, editors. *Monitoring and debugging of distributed real-time systems.* (IEEE Computing Society Press, 1995).

[33] S. Winter,P. Kacsuk. Software Engineering for Parallel Processing. in: *Proc. of the 8th Symp. on Microcomputer and Microprocessor Applications* (Budapest, 1994) 285-293.