Parallel program debugging, a case study at CERN

ir. René Schiefer

Mona Lisa is a new parallel programming paradigm, developed in the GP-MIMD project at CERN. One of the supporting tools for Mona Lisa is VIPER. It graphically visualizes the behavior of the parallel program using instant replay. The visualization is based on trace information, produced by the parallel program during (a previous) execution.
VIPER, combined with any commercially available sequential debugger, offers a debugging environment for Mona Lisa parallel programs. This debugging environment provides debugging at two levels: scenario replay with on-line visualization at the program level, and debugging of multiple threads using the sequential debugger at the sequential component level.
The development of VIPER has been defined as an OOTI project.

The GP-MIMD project

CERN[1] is an international laboratory near Geneva that performs particle physics research. This field of research is highly computational intensive. The complexity of the experiments performed at CERN is continuously increasing. A good example of this is the LHC experiment, for which the off-line computing needs are estimated to be three orders of magnitude larger than current LEP experiments. This is pushing the demands on I/O capacity and CPU power beyond the limits of present technology. A consequence of this is that CERN is investigating parallel computing as a possible solution to the increasing computing requirements.

Over the years, CERN has invested a lot in the development of dedicated software, which is mostly written in FORTRAN77. Not only is the total amount of this software rather large (millions of lines of code), but also the size of the individual programs is in some cases considerably large (hundreds of thousands of lines).

Because of this, CERN is investigating the possibility of parallelising their existing sequential programs, rather than rewriting them completely. Among other developments and collaborations with industrial partners, this research is done by taking part in ESPRIT[2] projects. GP-MIMD[3] is one of these ESPRIT projects. The main goal of the GP-MIMD project is to build a parallel machine and to run parallelised applications on this machine. The architecture will be discussed briefly below.

The parallel machine is based on T9000 Transputer technology. This means it has a multi-processor architecture with distributed memory. The platform in the GP-MIMD project will have 64 nodes. The processors communicate over links, equipped with 64K virtual channels. A direct communication path between non-adjacent processors is achieved by using C104 fast switching chips.

The CHORUS/MiX operating system runs directly on the Transputer hardware. CHORUS is a modular distributed real-time operating system. For the application, it provides two basic functions.

- The services for the allocation of processes to different processors.
- The IPC (inter process communication) services which implement a flexible, reliable message passing system between processes.

The CHORUS/MiX r3.2 operating system interface is a subset of UNIX.

The main physics application running on the GP-MIMD machine is the physics event reconstruction program of the CP-LEAR experiment. For the parallelization of this (originally sequential) code a new parallel programming paradigm has been developed, called Mona Lisa. Mona Lisa offers a

[1] CERN: Centre Européen pour la Recherche Nucleaire
[2] ESPRIT : European Strategic Programme for Research and Development in Information Technology
[3] GP-MIMD : General Purpose Multiple Instructions Multiple Data machines

simple, abstract, and optimizable way to design coarse grained parallel programs. The user interface is a set of six primitives which can be added to any existing sequential language. Hence, a Mona Lisa program is, for example, a FORTRAN listing with Mona Lisa statements embedded in it.

To make successful use of the Mona Lisa paradigm, a Mona Lisa development environment has been implemented. This environment supports the various steps which have to be performed to translate FORTRAN Mona Lisa source code into an executable program for the Transputer hardware.

However, the development of a parallel program adds another level of complexity in comparison with sequential coding. The behavior of a parallel program in terms of interactions between sequential parts is not straightforward. In particular, the sequential parts may interact in such a way that the outcome of the parallelized program is different from the outcome of the original sequential program.

VIPER[4] is the tool in the Mona Lisa development environment that closes the loop in the software development cycle; it is used in the process of analyzing, improving and debugging the parallel program. The VIPER tool provides the user with the data, needed to investigate the interaction between sequential parts of the program. The data is directly related to the Mona Lisa paradigm, thereby retaining the high abstraction level of the parallelization. VIPER graphically visualizes the behavior of the parallel program using instant replay. This means that the visualization is based on trace information, produced by the parallel program during execution.

The remainder of this article explains the context of the VIPER tool: the programming paradigm it supports, and the functionality it provides.

The Mona Lisa paradigm

This section briefly describes the Mona Lisa paradigm concept and some implementation details. A more thorough discussion can be found in [1].

The sequential components which together constitute the Mona Lisa parallel program, are called *modules*. The variable space in a module is split into two sub-spaces: the *global* variable space and the *local* variable space. A local variable has a normal scope, that is, as defined in the original programming language. A global variable is visible outside its normal scope: other modules can access this variable as well, in a restricted way.

A global variable (or, in the case of arrays, segments of a global variable) can be in two states, called *exposed* and *hidden*. Read and write operations on variables in other modules can only be performed when they are exposed. The same goes for the case of a segment of a global variable array. So, if a global variable within a module is hidden, its value cannot be read or changed by another module. Global variables can be moved between the two states explicitly by calling Mona Lisa primitives. This is the key in Mona Lisa to establish data synchronisation.

The Mona Lisa paradigm has the following characteristics.

1. The parallelism is data-synchronized, as opposed to control flow-synchronized. Instead of letting the user define the parallelism between modules explicitly using parallel language constructs (such as ALT in Occam), the parallelism is expected to be established in some way by the run-time environment. This is called *implicit parallelism*. The modules synchronize on the availability of data.

2. The communication between the modules is asynchronous and directed. Asynchronous communication allows flexible program design. Directed communication is necessary to get reasonable performance.

3. Startup, termination, and unloading of the parallel program is controlled by a central mechanism called the *program manager*. This avoids the trouble with dangling modules or errors due to an incorrect startup sequence. A dangling module is a module that is still running although the program itself has terminated. The program manager also detects deadlock and module abortion, in which case the parallel program is terminated with an appropriate error message.

To apply the Mona Lisa paradigm to a sequential program, two things have to be done.

[4]VIPER : VIsualization of Parallel Execution at Run-time

1. The program has to be partitioned into modules. Every module is effectively a sequential program by itself.
2. The data-flows between the modules have to be defined. This means defining the global variables, and putting Mona Lisa primitives in the code.

The Mona Lisa interface

Mona Lisa provides a set of six primitives, which enable the following.

1. Reading the value of a global variable (segment) in another module.
2. Changing the value of a global variable (segment) in another module.
3. Moving a global variable (segment) between the exposed and hidden states.

Table 1 lists the format of the Mona Lisa primitives (parameters are highlighted in italics), together with their operational semantics. The *var* parameter represents the global variable which is being manipulated. The *mod* parameter stands for the name of the module to which the global variable *var* belongs. The *buffer* parameter represents the variable to which the value of the global variable *var* is copied. Similarly, the *supply* parameter represents the variable whose value is copied to the global variable *var*.

The *rdglb* primitive is useful for making global data available to a collection of modules. The *inglb*/*inrglb* primitive, in combination with the *exposeglb* primitive, is useful for communications between two modules in a data-synchronised fashion. The *wrglb* primitive is provided to make an artificial RPC-mechanism possible: a client module calls a remote procedure by writing the id of the procedure in a global variable of the server module. The *hideglb* primitive is provided to enable undoing an *exposeglb*, although this is more for theoretical reasons then for practical use.

Mona Lisa implementation

A Mona Lisa program of N modules is implemented as a set of $2N + 2$ threads. Although the programmer regards Mona Lisa modules as single-threaded, they are in fact implemented as two threads, the module thread and the supervisor thread. The module thread runs the application code. The supervisor thread deals with the correct execution of Mona Lisa primitive calls. The two remaining threads of the parallel program constitute the Mona Lisa program manager. The roles of the supervisor and the program manager will be discussed shortly.

Primitive	**Description**
exposeglb(*var*)	Change state of *var* to exposed.
hideglb(*var*)	Change state of *var* to hidden.
mod.rdglb(*var,buffer*)	Read value of *var* from module *mod*.
mod.inglb(*var,buffer*)	Read value of *var* and change its state to hidden. *mod* is non-replicated module.
mod.inrglb(*var,buffer*)	Read value of *var* and change its state to hidden. *mod* is a replicated module. The instance which supplies the value is chosen non-deterministically.
mod.wrglb(*var,supply*)	Change value of *var* and change its state to hidden.

Table 1: Mona Lisa primitives

Supervisor

During program execution, different modules may try to access the same global variable. To ensure data-integrity of global variables, Mona Lisa primitives have to be implemented as atomic operations. For this purpose, every data manipulation (read, write, or move) is handled by a separate thread per module, called the *supervisor*. Both the supervisor thread and the module thread have access to the global variable space of the module. The data-integrity between the module thread and the supervisor thread is controlled via a binary semaphore. Actual data exchange between modules however is always done between supervisor threads.

Program manager

The program manager is an additional component of the parallel program, consisting of two threads. It basically performs four functions.

1. Startup of module threads at the beginning of the parallel program.

2. Termination of module threads at the termination of the parallel program or in a deadlock situation.
3. Monitoring of the state of the parallel program: abnormal termination because of module abortion, normal termination or deadlock.
4. Generation of a trace file containing all the trace messages. The trace messages will be explained further on.

One of the two threads is used solely for the trace file generation. It runs in the background of the other thread.

Traces

The implementation of the tasks of the program manager is based on traces. Whenever there is an important event in a module, such as the reception of data, a message is sent to the program manager which contains data like the sender of the data, the receiver, the primitive call to which message relates, etc. With these messages (traces), the program manager can trace the execution of the parallel program.

Functionality of VIPER

The visualization provided by VIPER serves three purposes.

1. Behavior analysis.
2. Performance improvement.
3. Debug facilitation.

Visualization involves large amounts of data. Therefore, abstraction, aggregation, and presentation of data is required at various levels.

- The parallel program level.
- The processor level.
- The process or module level.
- The task level.

The task level provides an intermediate level with respect to the other three. A task is simply a (user defined) set of processes. Tasks will reflect in most cases the way the software is developed: the entire program is first decomposed into logical units (tasks), and then each task is implemented as a set of processes.

Besides analysis, the trace information can also be used for optimization. The optimization concerns, for instance, the mapping of modules onto processors.

The usual way to debug a parallel program, is to look at the sequential parts (processes) individually. But in order to decide where to start the debugging activity, it is necessary to visualize the interaction between processes first. In particular, the programmer needs to know what kind of data is exchanged between the different processes. The VIPER tool supports the inspection of messages between processes. The next step then is to use a sequential debugger to debug a particular process.

The complete debugging environment as envisaged for the Mona Lisa paradigm provides the following functionality.

- Off-line visualization with VIPER, that is, by reading a trace file.
- On-line visualization with VIPER, that is, by processing traces directly as they are received from the program manager.
- Instant replay of a scenario, constructed from a (modified) trace file; VIPER makes sure that when the program runs, the modules interact according to the scenario. The current run can again be visualized on-line.
- Stepping through a single module using a suitable sequential debugger on the same platform as VIPER.
- Stepping through the primitive calls that are outside control of the modules being debugged by using VIPER.

The debugging environment is running on the SUN platform: VIPER, the sequential debugger, as well as a copy of every module and supervisor that has to be debugged. When debugging a Mona Lisa module, the user will actually look at the copy on the SUN platform. The advantage of this approach is that the parallel platform is not affected by the debugging activities. In the case of the GP-MIMD project, it also eliminates the need to write a FORTRAN debugger on the CHORUS platform.

A more precise description of the architecture can be found in [2].

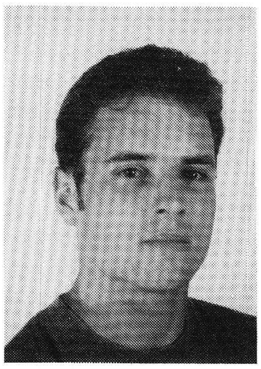
The development of VIPER has been defined as an OOTI-project. This was made possible by Dr. Peter van der Stok, member of the Faculty of Computing Science at the EUT and former CERN staff member. The project has now completed its design phase. VIPER has an object-oriented design, with C++ as implementation language. The graphics will be implemented with the C++ package InterViews.

References

[1] A. Schneider,
"Programming parallel machines"
CERN document, 1993

[2] René Schiefer,
"Architecture of a debugging environment",
CERN document, 1993

□



Ir. René Schiefer is student of the post-masters programme Software Technology. He is temporarily working in the GP-MIMD project at the ECP-division of CERN. He is a member of XOOTIC.

Agenda 1993

Project day IVO
Graduating students from all post-masters courses of Eindhoven University give short lectures on their projects.
Date: Wednesday September 22th, 1993, 10.00 - 15.30 h.
Place: Eindhoven University of Technology, Auditorium.

IVO graduation ceremony
26 students (eight of which are from OOTI) hope to receive their graduation certificate. Because of the large number of candidates, the ceremony has been split into two parts. Following on the ceremony, a drink will be held.
Date: Wednesday September 22th, 1993, 14.45 and 16.00 h.
Place: Eindhoven University of Technology, Auditorium.

Open day IVO
Information day for those interested to follow one of the post-masters programmes of Eindhoven University of Technology. In the morning, general information will be given, while in the afternoon the visitors will get specific information about the course of their interest.
Date: Thursday October 7th, 1993
Place: Eindhoven University of Technology.

OOTI Symposium
Afternoon symposium on the occasion of the fifth anniversary of OOTI. Title: "Specifying; a formality?". For further information, see elsewhere in this magazine.
Date: Thursday November 11th, 1993, 12.30 h.
Place: Eindhoven University of Technology, Auditorium.

VIE Excursion
Excursion to the Options Market in Amsterdam, followed by an evening with lectures, also in Amsterdam.
Date: Friday November 26th, 1993.
Place: Amsterdam.

IVO graduation ceremony
Students from various post-masters programmes in technological design of Eindhoven University will receive their graduation certificates.
Date: Wednesday December 8th, 1993, 16.00 h.
Place: Eindhoven University of Technology, Auditorium.