Using OMIS for On-line Monitoring in the GRADE Programming Environment*

Roland Wismüller
Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)
Technische Universität München, Germany
wismuell@in.tum.de

Gábor Dózsa
MTA SZTAKI, Computer and Automation Research Institute
Hungarian Academy of Sciences, Hungary
dozsa@sztaki.hu

Daniel Drótos
Department of Information Technology, University of Miskolc, Hungary
drdani@malacka.iit.uni-miskolc.hu

Abstract

To provide high-level graphical support for developing message passing programs, an integrated programming environment (GRADE) is being developed. GRADE currently provides tools to construct, execute, debug, monitor and visualise message-passing based parallel programs. GRADE offers the programmer an integrated graphical user interface during the whole life-cycle of program development and provides high-level graphical programming abstractions to construct parallel applications. The paper describes the integration of an OMIS compliant monitor system and GRADE in order to extend the programming environment with a more flexible monitoring support as well as to make possible the simultaneous usage of debugging and visualisation tools. OMIS is a recent specification of a universal on-line monitoring interface for parallel and distributed programs.

Keywords: Message-Passing, Visual Programming, Parallel Programming Environments, Monitoring, Debugging, Performance Visualisation

1 Introduction

The most widely used paradigm for implementing applications on distributed systems (including network of workstations and massively parallel computers) is the message-passing (MP) concept, and it is expected to be the most common approach for the next few years, too. The main reasons of its popularity are the simplicity of the concept and the fact that the message-passing paradigm closely corresponds to the way in which data are actually moved around in a distributed memory computer. Thus, MP libraries can be implemented very efficiently in such systems. Moreover, sequential languages (mainly C and FORTRAN) familiar to most application programmers can be used to perform the bulk of the computation.

Although there are quite a few MP system (e.g. PVM [5] , MPI [6]) which make it possible to execute parallel programs in distributed computing environments, the lack of real user-friendly support for development of such applications prevents most of the potential users from dealing with concurrent programming at all.

To cope with the extra complexity of parallel programs arising due to inter-process communication and synchronization, we have designed a visual programming environment called GRADE (Graphical Application Development Environment) in the framework of two Copernicus projects [20]. Its major goal

*This work is partly funded by German-Hungarian Intergovernmental WTZ / TéT project GRADE-OMIS, No. UNG-053-96 / D-62/96 and by the Hungarian Science Research Fund (OTKA) Contract Num.:F022105.

is to provide an easy-to-use, integrated set of programming tools for development of message-passing applications that can run either on real parallel computer or on heterogeneous workstation cluster.

Most important features of GRADE can be summarized as follows:

- All process management and inter-process communication activities are defined visually in the user's application. Graphics help in the better understanding the complex structure and run-time behavior of the distributed program even for users not familiar with parallel programming.
- Low-level details of the underlying message-passing system are hidden. GRADE generates all message-passing library calls automatically on the basis of the visual code. This approach has two basic advantages:
  - The programmer is not required to know the syntax of the MP library.
  - The same user application is able to run in different MP environments provided that GRADE can generate the code for those environments. Currently, GRADE can generate code only for PVM but MPI is to be supported very soon as well.

  However, local computations of the individual processes can be defined in C (or in FORTRAN in the future) independently from the visually supported message-passing related activities. Thus, GRADE provides a comfortable environment for parallelising existing sequential applications.
- Compilation and distribution of the executables are performed automatically in the heterogeneous environment.
- Debugging and monitoring information is related directly back to the user's graphical code during on-line debugging and post-mortem visualization of the run-time behavior of the application.

Monitoring the run-time behaviour of the distributed application is inevitable both for performance tuning and for correctness debugging. GRADE currently applies the Tape/PVM monitoring system [16] that generates a trace file during the execution of a PVM program for post-mortem analysis. However, in some cases this approach fails or becomes quite uncomfortable to use. For example, post-mortem tracefile analysis is not applicable to cyclic programs that usually run for very long time. Furthermore, generation of the Tape/PVM tracefile fails if the application terminates abnormally due to a run-time error. But in such a case trace analysis may be required in order to find the reason of the program crash. Additional drawbacks of Tape/PVM are that all processes must be monitored during the execution and that the ability to select the activities to be recorded is very limited. It would also be preferable to enable the use of the trace analysis tool during on-line debugging, e.g. when the application hits a global breakpoint [10] (i.e. to see what happened before the program has been stopped by the debugger). This is impossible with Tape/PVM as the tracefile is not available until the whole application has finished.

To overcome these problems, Tape/PVM is to be replaced by a monitoring system that supports more flexible observation of the application's run-time behviour. For networks of workstations running PVM, such a system is available with the OMIS compliant monitoring system OCM [22]. OMIS [14] is a recent specification of an on-line interface providing a portable way of monitoring and controlling parallel and distributed programs.

The rest of the paper is organised as follows. A brief description of the GRADE environment and the OMIS/OCM monitoring system is given in Sections 2 and 3, respectively. Issues about integration of OCM and GRADE are discussed in Section 4 followed by a comparison of our tools to related works. Finally, some conclusions are given.

2 The GRADE Graphical Programming Environment

GRADE currently consists of the following tools as main components:

- GRED: A graphical editor to design parallel applications [12]. The editor supports the syntax of the graphical language GRAPNEL (GRAphical Process NEt Language [11]).
- GRP2C: A code generator to produce the C+PVM code of the visual program.
- DDBG: A distributed debugger developed for PVM applications at the New University of Lisbon in Portugal [3].

- PROVE: A visualization tool to analyze and interpret Tape/PVM trace file information and present them to the programmer graphically.

A parallel architecture simulator developed at University of Barcelona [19] has also been integrated into an early version of GRADE but it cannot cooperate with the environment currently. (Hopefully, it will be adopted soon.)

Program development process in GRADE is described as follows. As a first step the user applies the GRED graphical editor to design and construct the parallel program written in a special visual programming language called GRAPNEL. Detailed description of the editor including an example application can be found in [12]. The visual programs are compiled to C+PVM code by the GRP2C code generator, which additionally creates other auxiliary files including makefiles used by the UNIX 'make' utility for building the executables. Code generation and compilation is fully automated on every host of the heterogeneous cluster of workstations. Having obtained the executables, the parallel program can be executed either in normal, debugging or trace mode. In debugging mode, the DDBG distributed debugger controls the execution of the program by providing commands to create breakpoints that can be attached either to visual elements of the GRAPNEL code or to specific C lines, step-by-step execution, animation, etc. In trace mode, a trace file is generated containing trace events about the execution of the various visual instruction of processes including message transfers. These events are visualised by the PROVE graphical visualisation tool assisting the user in spotting unintended message passing activities and performance bottlenecks in the GRAPNEL programs.

To illustrate current state and problems related to observation of run-time behaviour of GRAPNEL applications, we give a bit detailed information about monitoring and visualisation facilities of GRADE in the next subsection.

2.1 Monitoring and Visualisation in GRADE

GRADE provides the PROVE visualisation tool for the programmer to observe the run-time behaviour of his/her distributed application. Basically, two different types of two dimensional diagrams are offered by PROVE (where the horizontal axis is always the time axis):

- *Space-time diagram.* Each process is represented by a thick horizontal line. Every such line consists of sections coloured differently according to the actual state of the corresponding process (e.g. computing, sending a message, waiting for a message, etc.). Matching send and receive events are joined by arrows between process lines. Names of processes and that of hosts on which the processes were executed are shown in front of the corresponding line. An example diagram of this type is depicted in Fig. 1 (leftmost window). Time axis is represented at the bottom of the window as a thin line annotated by time stamps. Three process lines can be seen above the time axis. As it can be read in front of those lines, names of the corresponding processes are "Producer", "Buffer" and "Consumer"and they were executed on hosts "indigo.l"and "indy1.lp', respectively.

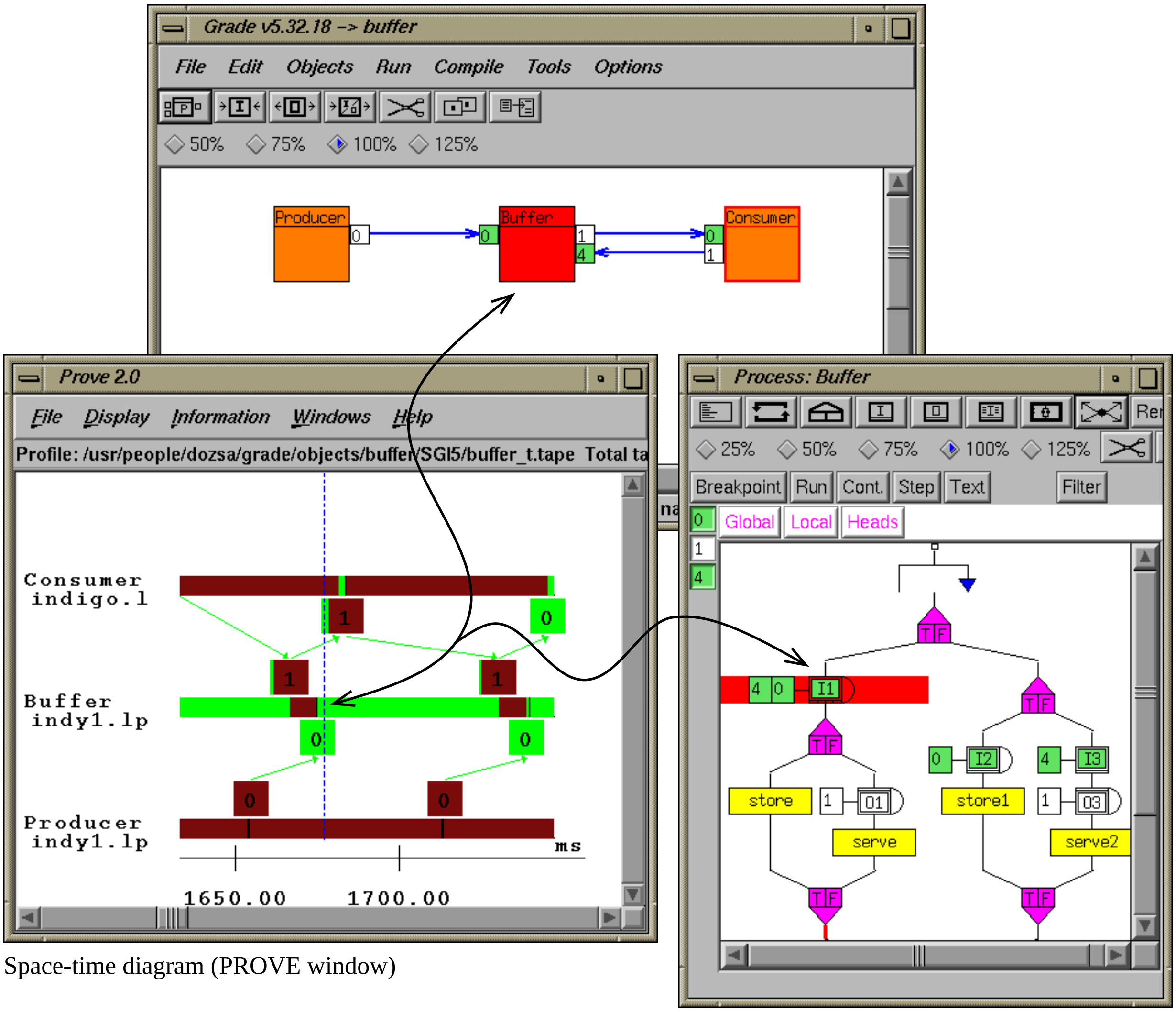
- *Process and Host Communication diagrams.* They display the amount of data transferred between various pair of processes or hosts, respectively, as simple two dimensional graphs (i.e. time vs. amount of data). They can be synchronised with the space-time diagram to show the same time interval which can be defined by the help of scroll and zoom mechanisms.

A unique feature of the environment is that a mutual highlighting facility between GRED and PROVE makes it possible for the user to identify the run-time activities of GRAPNEL icons in the PROVE windows and vice-versa to identify the responsible icon of GRAPNEL that generated a particular event of a PROVE window. This facility is illustrated in Fig. 1. A single mouse click on any point in a process line highlights the GRAPNEL icon that was active at that point of time in the process.

Currently, all graphs in the PROVE windows can only be drawn after the whole application has finished since GRADE applies the Tape/PVM monitoring tool to collect trace events upon which the visualisation is built. Tape/PVM has several advantages if only post-mortem visualisation is taken into account, e.g.:

- Precise computation of the global time stamps of events. Since global clock is usually not available in distributed systems, ordering of trace events is not a trivial task. Tape/PVM uses a non-intrusive, statistical method to estimate a precise global time reference based on some measurements performed before the application starting and after it has finished.

Process communication graph of the application (GRED window)



Space-time diagram (PROVE window)

Visual code of a process (GRED window)

Figure 1: Combined view of space-time diagram of PROVE and visual code in GRED windows

- Minimal perturbation of the user application being monitored. Trace events are stored locally in memory buffer of processes which are flushed to local files when necessary. Event records are compressed on-line to reduce the number of buffer flushes. The additional tasks used by Tape/PVM (for global control, clock synchronisation) are not active while the instrumented user application is running.

While post-mortem visualisation usually reflects the run-time behaviour of the monitored application in a much more precise way than on-line visualisation is able to do, there are some cases when the Tape/PVM approach fails or becomes too uncomfortable to use:

- Cyclic programs (e.g. distributed control systems). They usually work continuously so post-mortem visualisation of their run-time behaviour is not feasible.

- Applications that run for very long time (e.g. more hours or days). The size of the tracefile tends to be too large to be handled (i.e. to be stored to to be processed).

- Tape/PVM cannot generate tracefile if the program being monitored terminates abnormally due to a run-time error. Tape/PVM is not able to catch all types of exit events of processes because of the lack of capability to control the execution of the application. However, analysis of trace events can help much in finding the reason of a program crash especially if the run-time error is caused by unintended message transfer among processes.

- The possibility for the user to control which part of the application should be monitored is very limited in Tape/PVM. The main problem is that all processes must be monitored every time the instrumented application runs. However, in case of large number of processes it often happens that the programmer is interested in only a subset of processes to correct uncorrect run-time behaviour or to optimize a well defined part of the whole application. In such situations Tape/PVM inhibits the user from reducing the size of the tracefile – by filtering out all events to be generated by unrelevant processes – which could speed-up his/her work substantially.
- As Tape/PVM creates the global tracefile only after the whole application has finished, visualisation of run-time behaviour of the user's program is not possible in GRADE during on-line debugging although, PROVE's graphs could provide very valuable information in addition to that of the debugger (e.g. what messages have already arrived at a process stopped by the debugger, etc.).

To overcome these problems, the Tape/PVM monitor is to be replaced by OCM that supports more flexible observation of the run-time behviour of distributed applications.

3 The OMIS On-line Monitoring Infrastructure

OMIS is a recently developed specification of a universal on-line monitoring interface for parallel and distributed programs. Unlike other approaches, OMIS is not restricted to a single class of tools, but supports a variety of applications, including debugging, performance analysis and visualization. The detailed specification of the interface is published in [14][1]. We will, however, present two features of OMIS in more detail that are important for the following discussions.

A tool interacts with an OMIS compliant monitoring system by calling an interface function with the parameters specifying a service request and a callback function. Service requests consist of an event specification and an associated list of actions. The semantics of such a request is that the action list will be executed by the monitoring system each time the specified event is detected. If the event specification is empty, the action list is executed immediately and only once. A set of so-called *event context parameters* allows to pass additional information on the event occurrence to the actions. As an example, consider a debugger setting a breakpoint. The corresponding service request could look like this:

```
thread_reached_address([p_1,p_2], 0x1234): print([$proc]) thread_stop([$proc])
```

This means: whenever a thread in process `p_1` or process `p_2` reaches address `0x1234`, return the ID of that process and stop all its threads. When an action list returns some information to the tool, as is the case in our example, the specified callback will be invoked whenever a new result is available. Thus, by specifying a set of service requests, a tool can program the behavior of the monitoring system. It is important to notice that although OMIS offers centralized control over a distributed system, the event/action paradigm allows the specified actions to be migrated towards the places where the events will occur (i.e. the local node or even the monitored process). Thus, both response time and communication overhead can be reduced.

A second feature of OMIS is that the monitoring interface is extendible. Thus, tools can add their own events and actions to satisfy their needs. In the case of GRADE, OMIS already defines all the needed events (mainly the beginning and end of calls to the parallel programming library), but we may need an additional action for writing event information to a trace buffer. Once this action is available (let us call it `trc_add_event`), a tool can program the monitoring system to trace a set of events by sending a number of requests similar to the following one:

```
thread_has_started_lib_call([],"pvm_send"): trc_add_event(17, $node, $proc, [$par0,$par1])
```

This request will generate a trace entry with type tag `17` when any process (denoted by the empty list `[]`) started a call to `pvm_send`. The monitoring system will automatically take care of routing this request to all nodes, detecting the event, passing the event context parameters, and calling the action. As requests can be defined and deleted at any time, the tool has full dynamic control over the events and processes to be monitored. Of course, in addition to the `trc_add_event` action, we need another action that actually returns the event trace to the tool.

[1] See also `http://wwwbode.informatik.tu-muenchen.de/~omis`

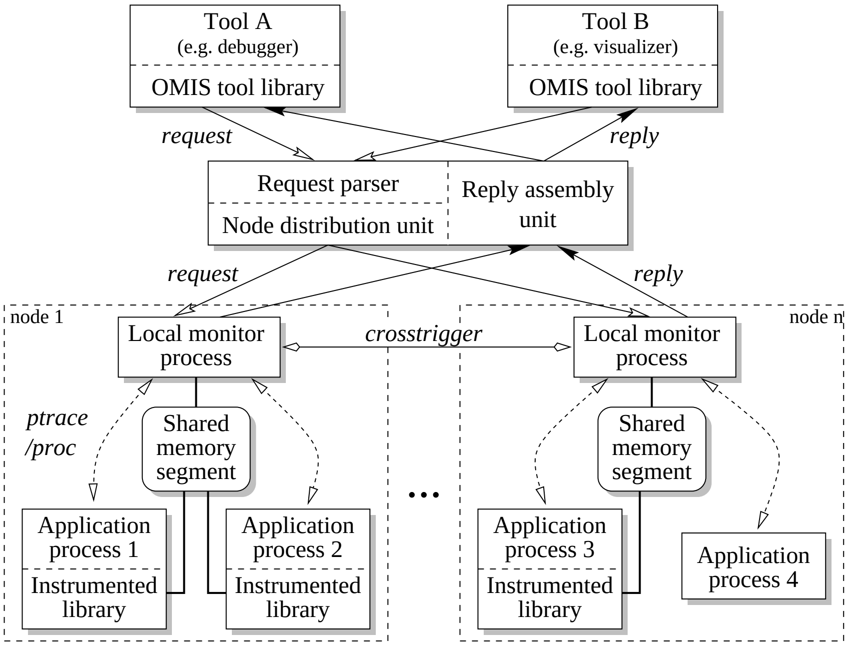


Figure 2: Coarse Structure of the OCM

In addition to specifying the interface, we have also implemented an OMIS compliant monitoring system (OCM) for PVM on networks of workstations. A port to MPI is currently done. The basic structure of the OCM is depicted in Fig. 2. We will only present a brief overview here; for a detailed description the reader is referred to [22].

The OCM basically consists of a monitor process on each node of the target system. Each process is responsible for processing incoming requests, retrieving local information, and detecting certain local events mostly related to debugging (e.g. an application process receives a signal).

In addition, there are monitoring components contained in instrumented libraries linked with the application processes. The task of these components is to detect events within the context of the triggering application process and to execute simple actions directly in that process. In this way, we avoid time-consuming context switches which would result in excessive perturbation. Typical examples for events being detected by these components are the beginning and the end of library calls. In order to tell these monitor components what they shall do, we use a shared memory segment that stores information on which events to monitor and which actions to execute. Note that actions in the application processes should not return results (except in the case of errors), since this would again result in context switches. However, they may update data stored in the shared memory segment (e.g. a trace buffer), which is read by actions executed in the monitor process upon request from a tool.

As OMIS allows a tool to issue requests that relate to multiple nodes, but each monitor process can only operate on its local node, we need an additional component, called Node Distribution Unit (NDU), that analyzes each request and splits it into pieces that can be processed locally. The NDU is also responsible for assembling partial results and returning them to the tool. Currently, the NDU is implemented as a central process, which communicates with the tools and the local monitor processes via message passing.

4 Using the OCM to Support On-line Visualization in GRADE

4.1 Requirements

To support on-line visualization in GRADE, we need to acquire a stream of relevant events occurring in the application program. As the visualization of this event stream will be on-line, eventually integrated with a debugger, there are several requirements for the monitoring infrastructure:

First, GRADE should be able to modify the set of events and/or processes to be traced at run-time, based on e.g. the currently visible displays. As outlined above, this requirement is already met by using the OMIS interface. In fact, a simple event trace can be already specified with the basic OCM, by issuing requests like

```
thread_has_started_lib_call([],"pvm_send"): print([17, $node, $proc, $par0, $par1])
```

This will result in a reply sent to the tool whenever the event occurs, very similar to the trace facility integrated into PVM 3.3.

However, this simple implementation conflicts with the requirement that the tracing should be reasonably efficient. As the main source of overhead is message passing, events should be buffered locally on each node and be transmitted to the tool only infrequently.

The need for buffering, on the other hand, interferes with our goal that visualization should be usable in combination with a debugger. There are two issues here: First, when the debugger stopped the program at some point in its execution, the visualization should be up-to-date. This requires a method to flush the event buffers on request of a tool. The second issue is that buffered events must not be lost when an application process crashes due to a bug. This means that the buffer must either be in a persistent memory (e.g. a shared memory segment) or it must be read before the process actually terminates. Up to now, these issues have not been addressed in event monitoring systems.

Finally, the event stream should reflect the actual order of the events as closely as possible in order to show a visualization consistent with the actual execution.

As we have outlined in Section 3, the OCM already provides the infrastructure for event detection, so generating a trace basically requires two actions to be defined, one for adding events to the trace, the other one for reading the trace.

4.2 Adding Events to a Trace

As we have seen, efficiency reasons dictate the use of in-memory trace buffering on each of the monitored nodes. There are three alternatives for implementing this buffering: As there are multiple processes on each node, we may either use a common trace buffer in the shared memory segment, or local trace buffers for each process. In the latter case, the buffer may either be in the shared memory or in local memory. Unfortunately, there is no clearly distinguished best alternative.

Using a common trace buffer for all processes on a node has the great advantage that the actual order of events is preserved in the buffer. However, a main disadvantage is that access to the buffer has to be synchronized. Using UNIX semaphores is prohibitive, as this would require two system calls per event, resulting in unacceptable perturbation. A suitable alternative can be spin locks, but still they result in a locking overhead that is strongly rising with the number of processes per node. However, a couple of processors offers atomic fetch-and-add instructions (e.g. `lock xadd` in the x86 processors) that can be used to synchronize access to the buffer without any overhead. This is due to the fact that we only have to append to the buffer. The fetch-and-add instruction allows to atomically get the pointer to the end of the buffer and to increase that pointer by the amount of space needed to store the event. Once the space is reserved in that way, the event can be stored without any further synchronization. The drawback of this method is, of course, that it is not usable for all target platforms.

We can also completely avoid the need for synchronization by using a local trace buffer for each process. However, in this case, the actual order of events cannot be fully preserved. Although each event is time-stamped, clock resolution is usually not high enough to avoid events in different processes having the same time stamp. When merging the local buffers, the order of these events can not be determined. Using sequence numbers is not a solution, since this would again require synchronization when incrementing the sequence counter.

Event buffers in the processes' local memory impose the additional problem that the buffer is lost when the process terminates. Thus, we need additional mechanisms to save the buffer. The main issue here is to detect that a process is going to terminate before it is actually deallocated. This can be achieved by using standard events defined by OMIS to monitor both signals and calls to the `exit` system routine. The problem does not occur when the buffers are allocated in the shared memory segment. However, most operating systems impose limits on both the size and the number of shared memory segments, thus restricting the scalability of this approach.

The OCM provides the infrastructure to easily implement and compare these different approaches. Taking all of the above aspects into account, we expect that a common event buffer for all processes on one node, together with atomic fetch-and-add operations or spin locks will be the best solution. The performance degradation of spin locks with high process numbers is not a severe problem, since on uniprocessor nodes we expect to have only a few processes. Multiprocessor nodes on the other hand will most likely be based on processors supporting enhanced synchronization primitives like fetch-and-add.

A remaining question will any buffering mechanism is how to handle a buffer overflow. Following the event/action paradigm of OMIS, our solution is to generate an event when the buffer is filled up to a certain limit (e.g. 80%). The tool can then associate any action with this event; typically it will add an action that reads the event stream (see next section) and thus empties the buffers. When the buffer is really full, any process trying to add an event will be blocked until the event stream has been read. Since there is still some buffer space when the "buffer full" event is generated, this situation should occur only infrequently.

4.3 Reading the Event Trace

As we want to support on-line visualization, it should be possible to read the event buffers at any time during the application's execution. Furthermore, since the visualization will be integrated with a debugger, we must be able to read the buffer independently of the application's state. E.g. some of its processes may be stopped at a breakpoint or may have crashed. This requires that the event buffers have to be read by the monitor processes, which are independent from the application. When the buffers are in the processes' local memory, we have to use operating system mechanisms to access them. On modern UNIX systems, which offer a `/proc` file system, this does not impose a performance problem, as data transfer over this interface is still an order of magnitude faster than network transfer needed to send the data to the tool.

Since one of our requirements defined above is that the event stream sent to the tool should reflect the actual event order as close as possible, it becomes necessary to merge the event streams obtained from local buffers to a global stream. In practice, there are two ways to achieve this: The order of two events in the global stream can be determined by comparing the events' time stamps, or by using causal dependencies between these events. Although the use of causal dependencies is the only way to guarantee that visualization will always be consistent, it is not possible to use this method in a general monitoring environment. The reason is that whether or not a causal dependency exists between two events heavily depends on the semantics of these events. The event recording mechanism, however, does not know anything about the semantics of the events. Furthermore, as has been outlined in [21], the kind of relationships that need to be considered even depends on the type of visualization done by the tool. Thus, sorting events according to causality constraints can only be done in a tool-specific extension to the OCM. There are already suitable algorithms for this kind of on-line event sorting (e.g. [7]), so we will not address this topic in more detail here.

The best effort possible in a general monitoring environment is to carefully synchronize the local clocks on all of the nodes and to use the events' time stamps for sorting. For the OCM, this means that we will provide an additional action for re-synchronizing the clocks. This action may be called regularly by a tool according to its needs on synchronization accuracy[2]. Clock synchronization is based on statistical methods to compute the offset and drift of local clocks, as described in [16]. For an on-line event recording, there are, however, some limitations of this technique: First, if a local clock is fast, we cannot simply put it back, since time should only increase. Rather, we have to set ahead all the other local clocks. Second, since clock drift can be measured only a-posteriori, we cannot truely correct for the drift. On a network of workstations, it is even questionable whether the clock drift can be measured at all, since message transmission times show a large variability, which reduces the accuracy of the measurement. However, allowing for clock drift is not mandatory in an on-line environment, since clock synchronization can be performed regularly, whereas in off-line environments it can only be performed at the beginning and the end of the application's execution.

5 Related Work

A number of other visual programming environments have been developed for parallel applications (e.g. TRAPPER [18], EDPEPPS [4], HENCE [2], PSEE [15], MOGUL [9]). Most of them are based upon the idea that nodes represent parallel computation, and arcs represent interactions (of some form) among nodes. The problem with the HENCE and PSEE approaches is that they force computations to be split into separate processes when communications occur or when branching decisions control communications

[2]E.g. performance analysis tools typically require much less accuracy than visualization tools.

(i.e. some kind of data-flow approach). This can result in complicated, awkward and large process communication graphs.

This problem does not arise in TRAPPER and EDPEPPS, which systems are quite close to GRADE concerning both purpose and functionality. However, GRADE provides visual instructions to define communication operations while TRAPPER and VPE apply graphics only to define the process communication graph. Visual code of GRAPNEL processes can help the programmer to better understand the communication structure of a parallel application in both the design and the correctness/performance debugging phase of program development.

Furthermore, GRADE is unique in the sense that a distributed debugger and a visualisation tool are fully integrated into the environment, i.e. debugging and visualisation information are related directly back to the graphical representation of the application.

Concerning monitoring systems, one can state that there already is a large number of systems for trace-collection, e.g. Tape/PVM [16] or the built-in PVM trace facility [13]. While they are well suited to support off-line event visualization tools, they do not support on-line environments for both debugging and visualization, as we have it with GRADE. There are two main reasons for this: First they do not allow the set of events to be monitored to be changed during run-time. Thus, a user cannot request a more detailed visualization when he finds some suspicious behavior with the debugger. Second, a process's event buffer is not saved when the process crashes. Thus, the most interesting events (i.e. the ones immediately before the crash) are lost[3].

There are also a few systems supporting on-line event monitoring, however, none of them supports our purposes. Falcon [7] provides on-line event monitoring and steering, and also addresses the problem of sorting event traces according to causal order. However, Falcon is primarily designed for shared memory architectures and does not provide the functionality needed by debugging tools. Another interesting approach is the Application Instrumentation System (AIS) [17] currently developed at IBM. It uses a dynamic instrumentation technique that allows to insert pieces of monitoring code into running application processes. This technique has originally been developed in the *dyninst* interface [8], however, *dyninst* allows to instrument local processes only, while AIS supports distributed systems. The main advantage of this system is that no special preparation of the application program is needed. But at the time of this writing, AIS is not yet fully specified and implemented. Since AIS is designed mainly for performance analysis, it is also not yet clear whether a debugger can be built on top of this interface.

6 Conclusion

The GRADE graphical programming environment helps to simplify the task of designing and understanding parallel programs. An important prerequisite for this is GRADE's ability to visualize and debug programs at the design level. We have shown that for an on-line visualization, current event tracing libraries are not sufficient. We therefore need a more flexible monitoring infrastructure, such as the OMIS compliant monitoring system OCM. An analysis of the requirements and constraints imposed by on-line visualization has shown that monitoring support must differ considerably from current approaches, which support off-line tools only. An outline of an implementation of on-line tracing has been presented. We expect a first integration with GRADE to be available by the end of the year.

References

[1] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Recent Enhancements to PVM. *Intl. Journal of Supercomputing Applications and High Performance Computing*, 9(2), 1995.

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

[3] J. C. Cunha, J. Lourenco, and T. Antao. A Debugging Engine for Parallel and Distributed Environment. In *Proc. of 1st Austrian-Hungarian Workshop on Distributed and Parallel Systems, Miskolc, Hungary, October 2-4*, pages 111–118, 1996.

[4] T. Delaitre, G. Ribeiro-Justo, F. Spies, and S. Winter. A graphical toolset for simulation modelling of parallel systems. *Parallel Computing*, 22(13):1823–1836, 1997.

[3] This is not true for the tracing in PVM 3.3, as there is no buffering. However, this leads to enormous program perturbation (see [1]).

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

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

[7] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, and J. Vetter. Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs. Technical Report GIT-CC-94-21, College of Computing, Ceorgia Institute of Technology, Atlanta, GA, USA, 1994.

[8] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic Program Instrumentation for Scalable Performance Tools. In *1994 Scalable High-Performance Computing Conference*, pages 841–850, Knoxville, TN, Mai 1994.

[9] P. Kacsuk. Graphical environments to support parallel program development. In *Proc. of the 2nd Int. Conf. on Parallel Processing and Applied Mathematics*, pages 66–80, Zakopane, 1997.

[10] P. Kacsuk. Macrostep-by-macrostep debugging of message passing parallel programs. Submitted to IASTED PDCN'98 Conference, Las Vegas, 1998.

[11] P. Kacsuk, G. Dózsa, and T. Fadgyas. Designing Parallel Programs by the Graphical Language GRAPNEL. *Microprocessing and Microprogramming*, 41:625–643, 1996.

[12] P. Kacsuk, G. Dozsa, T. Fadgyas, and R. Lovas. The gred graphical editor for the grade parallel program development environment. In *Proc. of HPCN'98: High-Performance Computing and Networking*, pages 728–737, Amsterdam, The Netherlands, April 1998.

[13] J. A. Kohl and G. A. Geist. The PVM 3.4 Tracing Facility and XPVM 1.1. In *Proc. 29th Hawaii Intl. Conf. on System Sciences (HICCS-29), Heterogeneous Processing Minitrack*, Maui, Hawaii, Jan. 1996.

[14] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. *OMIS — On-line Monitoring Interface Specification (Version 2.0)*, volume 9 of *LRR-TUM Research Report Series*. Shaker Verlag, Aachen, Germany, 1997. ISBN 3-8265-3035-7.

[15] E. Luque, R. Suppi, and J. Sorribes. Overview and New Trend on PSEE (Parallel System Evaluation). *IEEE Software*, 1992.

[16] E. Maillet. Issues in Performance Tracing with Tape-PVM. In J. Dongarra et al., editors, *Proc. Second European PVM User's Group Meeting, Lyon*, pages 143–148. Hermes, 1995.

[17] D. M. Pase. *An API for Run-Time Instrumentation of Single- and Multi-Process Applications: Class Reference Manual.* IBM Corporation, 522 South Road, MS P-963, Poughkeepsie, New York 12601, May 1998.

[18] 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.

[19] R. Suppi et al. Simulation in Parallel Software Design. In *Proc. of Euro-PDS'97*, pages 51–59, Barcelona, 1997.

[20] S. Winter and P. Kacsuk. Software Engineering for Parallel Processing. In *Proc. of 8th Symp. on Microcomputer and Microprocessor Applications*, pages 285–293, Budapest, 1994.

[21] R. Wismüller. State Based Visualization of PVM Applications. In A. Bode, J. Dongarra, T. Ludwig, and V. Sunderam, editors, *Parallel Virtual Machine – EuroPVM'96*, volume 1156 of *Lecture Notes in Computer Science*, pages 91–99, Munich, Germany, Oct. 1996. Springer Verlag.

[22] R. Wismüller, J. Trinitis, and T. Ludwig. OCM — A Monitoring System for Interoperable Tools. In *Proc. 2nd SIGMETRICS Symposium on Parallel and Distributed Tools SPDT'98*, Welches, OR, USA, Aug. 1998. ACM Press. To appear.