EMU - Event Monitoring Utility Technical Report*

Siegfried Grabner, Dieter Kranzlmüller, Jens Volkert
GUP Linz
Johannes Kepler University Linz
Altenbergerstr. 69, A-4040 Linz, AUSTRIA/EUROPE
Phone: +43 (732) 2468 - [884 | 9499], Fax: +43 (732) 2468 - 9496
[grabner | kranzlmueller]@gup.uni-linz.ac.at

*) Deliverable D7V-1 of project CEI-PACT, subproject Work Package 7: "Execution Environment for Parallel Programs", supported by the Austrian Ministry of Science and Research (BMWF), project no. GZ 308.927.

Abstract

Supercomputing power is a solution to the users' demand for more accurate results and solving larger problem sizes. First problems arise at the design stage of the parallel software engineering process, where the user has to cope with the problem of distinct, concurrently executing processors. Whereas memory hot spots and bus contention are problems with shared memory architectures, nondeterminism arising from race conditions and the lack of a global clock are severe problems in context with distributed memory machines.

These problems continue at the implementation stage and also in the field of testing and debugging. Due to errors which depend on the message passing paradigm, the debugging process has to be extended. In this paper we will discuss our approach to error detection of concurrent events in distributed memory machines. The starting point for our approach is the event monitoring utility EMU, which traces a running program to gather information for the post-mortem analysis.

The main goal of the monitor EMU is to produce minimal interference to a running program due to monitoring overhead but also to provide all necessary data for the later analysis. The monitor is implemented on a nCUBE 2 distributed memory machine.

Contents

1. Introduction

Program development includes several steps to ensure the quality of the resulting implementation. The most important fact is a clear and structured design. This is the basis for the implementation work, which can be done by using computer aided tools. Another part to provide high quality software comes from the testing and debugging phase. Debugging tools are necessary to achieve error free programs.

These phases of the software lifecycle apply to the sequential as well as the parallel programming. Parallel programs differ from sequential ones because of multiple threads of control and necessary communication and synchronization established between these threads. Consequently, debugging parallel programs is more complicated than debugging sequential ones. The development and implementation of parallel debuggers is a must for further improvements in the quality of parallel programs due to the minimization of programming errors.

The features of such a debugging tool must be determined by all prospective errors. The first kind of errors may appear within program blocks on any processor between two communication statements. These blocks are processed sequentially and therefore local to one processor. The error detection can be done by using conventional sequential debugging methods like setting breakpoints and watchpoints or by single stepping the program, respectively.

On the other hand these errors may spread among other processors through the communication messages. This may produce succeeding errors on a group of parallel processors. The localization of these errors follows the idea of finding the first error which caused all others. To achieve a solution the dependencies of the threads due to the communication messages must be investigated. Then it should be possible to step back until a block is reached where the first error occurs. Within this block conventional debugging techniques can be used again.

The third kind of errors may happen within the communication itself. The result is a wrong communication structure or wrong transmission of data. There are many errors which may appear due to incorrect communication. Other errors are race conditions, deadlocks and stampede effects. The detection of these errors can only be done by investigating the communication structure of the program.

These errors and the problems with debugging parallel programs lead us to our debugging approach. With our toolset we want to help the user to find most kind of errors of parallel programs. The basic concept of our tools is the visualization of a global communication graph. This graph uses events which were recorded during an actual program run to establish a connection between concurrent processors on the basis of corresponding sends and receives. With the eventgraph the user has the ability to find most errors of parallel programs by inspecting or manipulating the communication events. Through trace driven simulation the detection of race conditions will be possible.

A main idea of our debugging strategy is the connection between communication events and the source code. This connection is established with the integration of a file browser in the toolset and allows the user to change the source as soon as an error is detected.

This technical report has the intention to present the first part of our debugging tools - EMU, the Event Monitoring Utility. This monitoring tool is used to record the communication events during a program run in tracefiles. The paper is organized as follows. First we introduce the purpose of our monitor as one basic part of our debugging strategy. A main part is the description of the most possible errors in parallel programs. Section 3 describes the facilities of the monitor with all its parts and the relation in between. The next section deals with events and their attributes and describes the recorded data. Section 5 shows our strategies to minimize the monitoring overhead and a comparison between the different strategies. After the description of our implementations we conclude with a short summary of the tool and the future aspects.

2. The Debugging Approach

Debugging parallel programs is more complicated than debugging sequential ones. The lack of a reasonable debugging strategy which is easy to use for programmers can be explained as follows:

- Race conditions may occur whenever two computations are allowed to progress in parallel. This may lead to nondeterminism in program flow. The occurrence of race conditions depends on each CPU's load, on the amount of network traffic and on the nondeterminism inherent in the communication medium. One consequence is the "irreproducibility effect" [SnHo88]. This means that running a program twice with the same input may yield different results. Therefore it is extremely difficult or even impossible to reproduce such an error in successive program runs.
- "Deadlock effect": if processors lock each other.
- "Stampede effect": if one processor runs into an error it is impossible to stop all other processors.
- The nonexistence of a synchronized clock leads to difficulties in determining the precise order of events occurring on distinct, concurrently executing processors. This includes that there exists no global state.
- The instrumentation of diagnostic tools can change the program behaviour strongly. This is the so called "Probe effect" [McHe89] or "Heisenberg effect" [SnHo88]. This effect may also occur with monoprocessor debugging. It can happen that original bugs vanish when program optimization is turned off [ApMc88].

- Another important factor is the number of processors used in a program run. The complexity of the debugging process increases if errors may occur on more than one processor.

The development of parallel programs is mostly based on serial algorithms. New inherently parallel algorithms are only a minority. The reason for this is that serial algorithms are already investigated in detail and mathematically proved correct.

The parallelization of serial algorithms includes a step for adding communication parts. These parts are most error-prone. On the one hand wrong message types, incorrect source or destination specifications of processors and wrong message lengths cause incorrect results. On the other hand race conditions may be introduced.

The basic part of our debugging approach is a trace of events. This trace is recorded with our monitor during a program run [GrKr94]. Its format is similar to the format of PICL tracefiles (Portable Instrumented Communication Library) [Geis90].

An event is defined as an action without duration that takes place at a specific point in time and changes the state of a process [RiTo92]. The most important events in our context are the communication events.

With these events stored in the tracefiles we generate an eventgraph. An eventgraph is a partially ordered graph (in time) that represents the interprocess dependencies between processors.

The graph will be visualized within ATEMPT - (A Tool for Event Manipulation) [GrKr95]. This tool allows the inspection of the eventgraph and provides facilities to detect the primary errors as specified above.

The next step is the possible manipulation of the eventgraph at places where race conditions may occur. The resulting eventgraph is used as a constraint for the simulator on the multiprocessor called PARASIT, which performs trace driven simulation [MiCh88, LeMe87].

In the simulation the user has the possibility to stop at points of interest like in traditional breakpoint debugging. There variables, message contents, and message queues can be inspected. The main idea of using a simulator is the detection of race conditions. This can be achieved by comparing the original tracefiles with the trace generated by the simulator. Another comparison can be done with the resulting outputs of the programs. If the tracefiles or the outputs are different, a race condition is located. An overview of our method is shown in figure 1.

The visualization of the eventgraph has further benefits. One is the possibility to place cuts in a program. Cuts are similar to breakpoints in traditional debugging with the difference that cuts are placed on more than one processor.

These cuts can be specified graphically in the eventgraph and are then automatically integrated in the source code. After another program run the data recorded at these cuts can be used for several investigations. A programmer can inspect variables

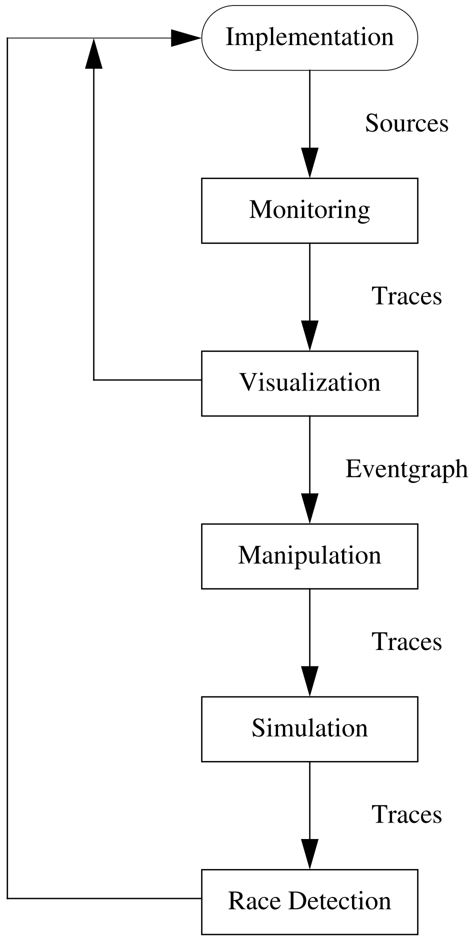

Figure 1. Debugging Overview

or message contents and other message parameters. It is also possible to examine the message queue of a specific processor.

Another benefit of cuts is the possibility to start at a certain cut and run only parts of the program. This is achieved by a mechanism similar to checkpointing in some operating systems. As a result the simulated program run can be shortened when an error is located in a certain code region. The concept of cuts has not been implemented yet and will therefore not be discussed in this paper.

The eventgraph can also be used to "vertically" debug one or more processors. This means that only the necessary surroundings in communication for one or more processor(s) will be simulated. This single processor (or the group of processors) can then be investigated in detail.

With our approach it is possible to determine a large group of errors in parallel programs. There are two kinds of program faults that can be distinguished:

- local errors
- global errors

Errors that appear within a code block that resides completely on one single processor are local errors. They do not influence any other processor's behaviour through communication. Because of the locality of such faults a traditional debugger can be used to localize and correct them.

The more difficult errors are the global errors. Although these mistakes may also appear only on one single processor they influence the behaviour of other processors. The influenced processors are all communication partners of the processor with the faulty program code. The simplest errors in this case are wrong communication structures and wrong data that are sent to another processor.

The worst errors that may occur are known as race conditions in literature. A data race is a typical case of unintended nondeterminism [ChMi91]. The problem with this kind of bug is that it appears sporadically. Therefore it cannot be localized and corrected easily. Several strategies for the detection of race conditions have been investigated [ChSt91, ChMi91, LeMe87, MiCh88]. A widely known technique for the investigations of race conditions is trace driven simulation [ChSt91] with instant replay [LeMe87]. Our approach tries to combine these techniques within an integrated debugging environment.

3. EMU - The Event Monitoring Utility

The first step in the debugging environment is our monitor EMU. It is responsible for the collection of all necessary eventdata of a running program. This is the basis for the analysis tool and the trace driven simulation. There exist several strategies for monitoring [Joyc87] with two goals: error detection and performance evaluation.

The main goal of a monitoring tool is *to cause very little or even no timing overhead* [Mari90]. Another goal which depends on the one specified above is that a monitor should not change the system behaviour of the instrumented program. These goals are reached with three different types of monitors:

- hardware monitors
- software monitors
- hybrid monitors

The hardware monitors are restricted to a few observation points and only provide a low-level interface as well as low-level data. The advantage of this kind of monitors

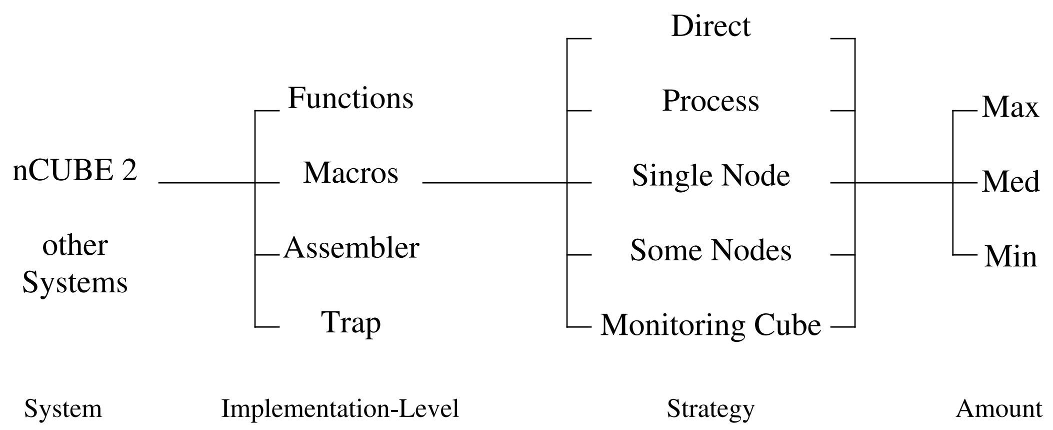

Figure 2. Monitoring Scheme

is the very little intrusiveness in programs. The software monitors produce some degree of interference in both time and space of the monitored program, but can present the information in an application oriented manner. A mixture of hardware and software monitoring concepts are hybrid monitors. The intention of this kind of monitors is to keep the influence of the monitor as little as possible.

Our approach to monitoring can be seen as a software monitor with aspects of hybrid monitoring. The tool is implemented totally in software and uses knowledge about the architecture of the monitored system. In the first implementation this is an nCUBE 2 distributed memory supercomputer [nCUBE92a] and the monitor includes certain constructs especially for its hypercube topology [nCUBE92b].

Another feature of our monitor is the conversion of its tracefile format to other formats like PICL (Portable Instrumented Communication Library) [Geis90]. This allows us to use other tools like ParaGraph [HeFi93] as well as our tools for analysis of tracefiles recorded with PICL. ParaGraph is a tool that visualizes the behaviour of parallel programs with a main look at the obtained performance.

Our tracefiles contain information about the following events:

- Communication-Events: Send, Receive, Test
- Global Events - Concatenated Events: Synchronize, Global-Sum, Global-Max
- Trace Events: Queue Inspection, Variable Inspection

Each event is a complex structure with information about the program like supposed and actual (communication) parameters, the source code reference and some timing parameters. The amount of collected data can be specified between a minimum of essential parameters and a maximum with all useful information. The structure of our monitor is shown in Figure 2.

The monitoring scheme shows several distinctions between systems, implementation-levels, strategies and the amount of data that has to be recorded. The first distinction concerning systems shows that the monitor is designed in a way that should allow porting it to other platforms. The implementation influence is visible through the implementation-level which describes the distance from the hardware to high-level programming languages like C. These levels also express the usability of tools for the intended purpose of the monitor.

The following five strategies show different implementations of the monitor with various advantages and disadvantages:

- Direct: Monitor is fully included in the computing program
- Process: A separate task performs the monitoring on the computing nodes

- Single Node: One node acts as monitoring slave and collects the monitoring data of all computing nodes.
- Some Nodes: A few nodes act as monitoring slaves. This strategy is used especially in hypercube architectures that allow subcube allocation.
- Monitoring Cube: A separate subcube provides one monitoring slave for each computing node.

The amount-level determines the amount of traced data and can be described as follows:

- Maximum: exhaustive timing
- Medium: simple time stamps
- Minimum: *logical clocks* as proposed in [Fidg88]

Exhaustive timing measures monitoring overhead, communication duration, and the time of event occurrence whereas simple time stamps only indicate when a communication event happened. The difference to the logical clock approach is that here absolute timings are considered whereas with logical clocks only event numbers are recorded.

This complex monitoring scheme was designed to meet the goals of a monitoring tool, which tries to achieve a minimum of

- timing overhead
- influence on traffic
- memory consumption

The minimal monitoring overhead can be achieved at the trap-level with minimum amount of tracedata. Further improvements can be made in using the monitoring

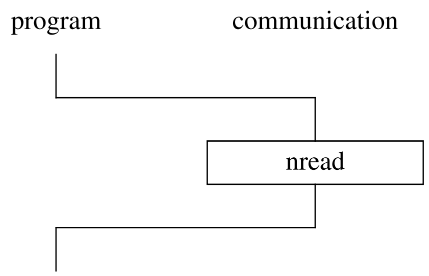

Figure 3a. Standard Communication

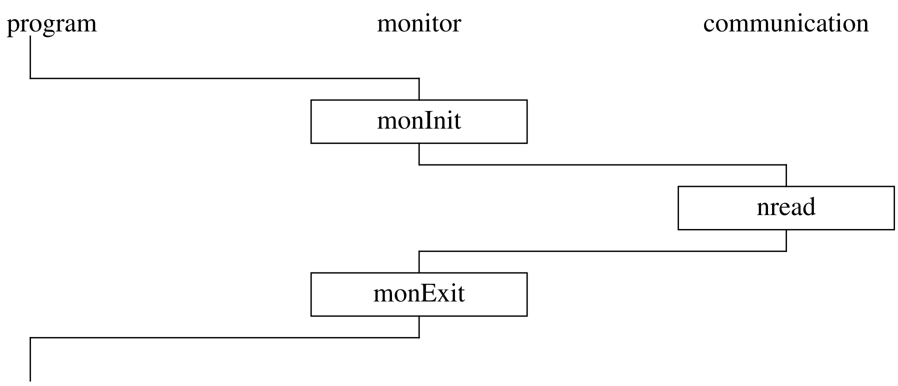

Figure 3b. Monitored Communication

cube strategy. A disadvantage of this strategy is the loss of half the allocated cube which acts as the monitoring cube. The main advantage is that the monitoring overhead on the computing nodes becomes a minimum and the network traffic between computing nodes and monitoring nodes has no influence on the network traffic that concerns computation. The reason for this is that the monitoring slaves can be positioned in a way that computing nodes and monitoring nodes use a separate subcube [GrVo93].

4. Monitoring Overhead and Event-Attributes

4.1 Events

To determine the monitoring overhead, it is useful to know how monitoring affects the standard communication. In standard communication the program switches to the message passing interface whenever communication routines are called. After performing the appropriate library subroutine, the communication returns to the calling program (see figure 3a). The monitor has to provide the same functionality to the program, but must also generate the needed event attributes.

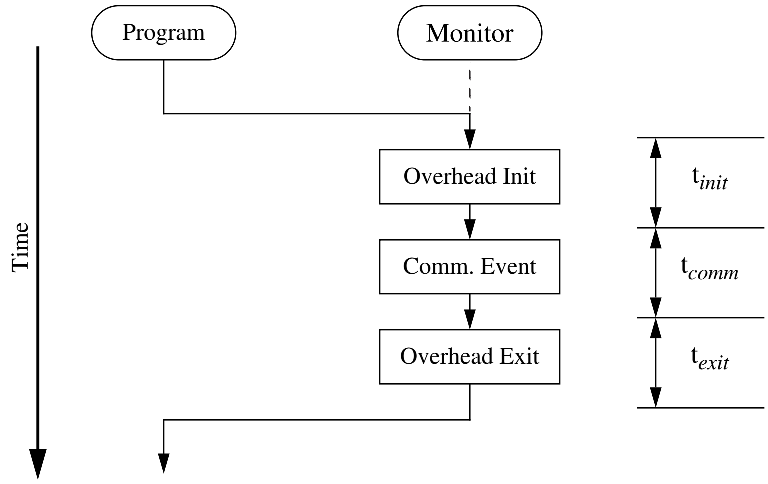

Figure 4. Monitoring Overhead

This leads us to monitored communication, where all communication routine calls are changed to special monitor routine calls. These calls provide the functionality needed for the monitor and a call to the original communication routine (see figure 3b).

In our monitoring approach, the monitoring functionality is divided into two parts, *monInit* and *monExit*. The first part performs monitoring activities at the initialization of each communication event. If an error occurs in the communication routine and no information is generated at the initialization time, it would not be possible to trace the error exactly to this communication statement. With error-monitoring enabled this information is stored in the files immediatly to save data for the analysis tools.

The second part - *monExit* - performs the additional monitoring activities at the end of the communication. This provides the ability to know when a communication statement is completed.

The size of the overhead mainly depends on the following parts:

- amount of eventdata produced
- integration of the monitoring process in the computing process
- level of implementation.

As specified above these points produce monitoring overhead in terms of memory and time consumption at the parts *monInit* and *monExit*. Figure 4 shows the three parts of the monitor with the times t_{init}, t_{comm} and t_{exit}. t_{init} is the time required by the monitor at the initialization phase of the communication. This part is essential for error-monitoring facilities, where the knowledge about running (not finished) communication events is important. The communication time t_{comm} is the required time for the transmission of a message. After the communication some monitor-overhead t_{exit} exists for the completion of the current event. Table 1 specifies the tasks of the monitor at the initialization and exit phase for the two different strategies *direct* and *monitoring cube*.

Table 1: Monitoring Overhead

Overhead	direct	monitoring cube
Init	- collect data - measure time - create Communication-Event	- collect data - send data to Monitor-Node
Exit	- measure time - save Communication-Event	- send data to Monitor-Node

The *init* and *exit* parts are constant in time due to a constant number of operations, which have to be performed with each event. The difference between the two monitoring strategies is mainly the creation and saving of communication events. The strategy *direct* uses file access operations and therefore costs a lot of time. With the monitoring cube most overhead is produced on the monitoring slaves and therefore the only necessary overhead is to send the data to the slave nodes.

The time for the communication is constant due to the time of transmission of the message but influenced by traffic conditions for send and receive events. Send events are delayed when the path is blocking whereas receive events are delayed when a message is not arrived completely. We assume that interrupts are used to determine the availability of outgoing channels and therefore minimize the blocking influence at the sending nodes. This means that messages are stored in a special memory called message buffer and the start of the transmission is handled through an interrupt service routine, which is called as soon as the required channel is free. As soon as the source code of the communication routines will be available to us we will take a closer look at this behaviour.

To determine the blocking time at the receive events we have to consider the following formula for the measurable communication time:

$$t_{comm} = t_{trans} + t_{block}$$

The time t_{comm} is the time needed for the original communication event, t_{trans} is the time for transmitting the message in the communication network and t_{block} is the blocking time. The time for the transmission of *n* bytes can be considered to be

$$t_{trans} = t_{startup} + n \times t_{send}$$

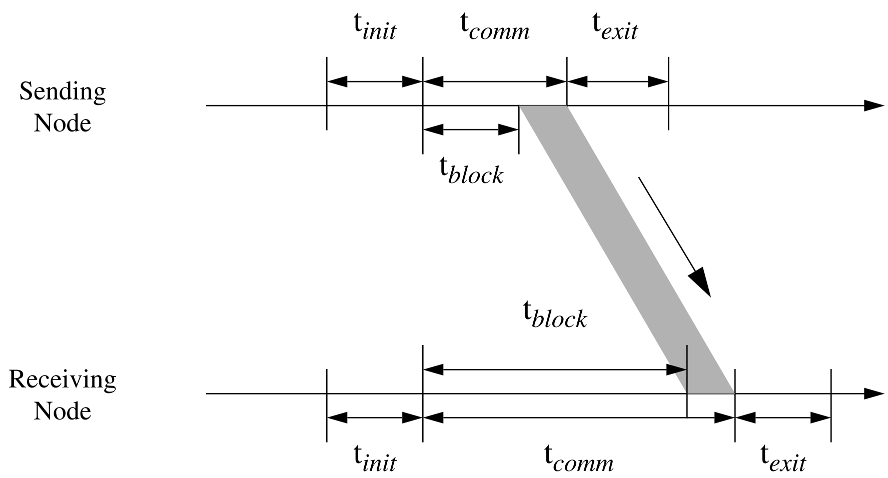

Figure 5. Timing Overhead

where $t_{startup}$ is the startup and network latency and t_{send} is the time used to transmit one byte at the sender or receiver. This means that the time t_{trans} for a given number of bytes is nearly constant. The time t_{comm} can be evaluated using the following equation

$$t_{event} = t_{init} + t_{comm} + t_{exit}$$

The times t_{init} and t_{exit} are constant as described above and the time t_{event} is obtained during the monitoring. Concerning the facts above we are able to calculate the time for t_{block} which is a measure for the network traffic influencing the current event. The relations between the different timings are shown in Figure 5.

4.2 Amount of Eventdata

The amount of eventdata, which has to be produced, depends on the information which is needed for the post-mortem analysis tool. In our approach we select the amount of recorded trace data by choosing the appropriate amount level. We define three levels for *minimum*, *medium* and *maximum* data recording. The produced data of these levels can be obtained from table 2.

The lines for *monInit* and *monExit* determine the initialization and exit activities of the monitor. The whole table has to be interpreted cumulative, which means that the *maximum* level contains all activities of the *medium* level, which again contains all activities of the *minimum* level.

Table 2: Amount Level

Level	total	eventdata
Min	monInit	EventType EventName
	monExit	Logical Clock of Event Logical Clock of Partner (if available) Accepted Communication Partner Accepted Message Type Accepted Message Length
Med	monInit	Timestamp before Communication
	monExit	-
Max	monInit	Supposed Communication Partner Supposed Message Type Supposed Message Length
	monExit	TimeStamp after Communication

4.2.1 Minimum Level of Amount

The *Minimum Amount Level* of the monitor produces only a minimum set of event attributes which is necessary for further analysis. At the initialization it is obvious to record the type of event that currently happens and the name of the event. The difference between the type and the name of an event is, that different send events with different event names share the same event type *send*. The identification of an event is done with three symbols: Event-Type, Subtype and Eventname.

The most important event types and the corresponding event names are shown in table 3.

The communication events are *receive*, *send* and *test*. The global events or other communication events are synchronization events and global arithmetic events like global summation. Trace events are source code connections, variable and message queue inspection and events for the control of the trace-file generation.

This first identification of the events determines the rest of the eventdata, which are stored after the identification header. In this paper we will not present further information about the Global/Others or Trace events, because these events are not important for the main goal of error-detection. A summary of all produced data will be available in the system documentation.

As we have already mentioned the most important events are the communication events. After the initialization phase the original communication is performed. The results of these function calls must be stored in the tracefiles to obtain information

Table 3: Event Types

Event Type	Sub-Type	Name
Communication	Receive	nread
		nreadp
	Send	nwrite
		nwritep
	Test	ntest
Global/Others	Synchronization	nsync
	Summation	nsum
Trace	SourceInfo	
	Variable Inspection	
	Message Queue Inspection	
	TestLoop	
	TraceFlush	
	TraceOn/Off	

about the communication. This information is only available at the exit-part of the monitoring function and consists of

- logical clocks and
- the accepted parameter packet.

The logical clocks are a simple form of the logical clock concept that was proposed in [Fidg88]. It is a standard strategy to establish connections between events from different processors without having a synchronized global clock. There are two values generated to produce correct connections: logical clock number of the event and logical clock of partner-event (send-receive). The logical clock value of each event is an integer number beginning with zero and increasing at each event. The logical clock of the partner is the value generated by the corresponding partner event on the other processor. This means that a receive event stores its own logical clock value and the logical clock value of the corresponding send event.

The accepted parameter packet is a data-structure which contains information about the actual performed communication statement. The values are the values of the actual transmitted or received values. These values are

- communication partner
- message type
- message length.

These values do not need to be constants at the initialization phase. For instance a receive command can allow communication from an unknown partner with unknown type and unknown message length. The recorded data for such a receive command consists then of the actual values from which partner the message was received and which message-type was used.

With this set of eventdata it is possible to visualize the eventgraph. Inspections of the accepted parameter packet allows the user to check, if correct messages were received. The missing part in this eventgraph is the actual time between two events. The user has no possibility to check whether the time between two events is a second or an hour. To overcome this problem the next model produces simple time-stamps.

4.2.2 Medium data amount

The medium level includes all functionality of the minimum level and extends the generated eventdata with simple time-stamps. These time-stamps include the value of the local clock of each processor. The generation of the time-stamps is done at the initialization phase. This gives the possibility of setting timeout values for hanging communication statements.

The time-stamps allow the evaluation of the computing time between two communication events, if these events are happening on one processor. Since the time generated on one processor is not the same time generated on another processor, this time-stamps allow now global timing for all processors, but only for local events on one processor.

4.2.3 Maximal data acquisition

At the maximum level of data amount the monitor generates all currently useable information for the analysis of parallel programs within our debugging environment. It includes all functionality of the levels *minimum* and *medium* and generates the following additional eventdata:

- supposed parameter packet
- additional time-stamps

The supposed parameter packet is the extension to the accepted parameter packet. It consists of the communication parameters, which are supposed to happen in the next communication event. The actual resulting parameters are stored in the accepted parameter packet. As these two packets share the same data-structures, they are combined in a communication parameter packet structure. The comparison between these two packets allows the user to check, if the communication happened with the supposed parameters.

The additional time-stamps (after communcation) are mostly needed for performance evaluation. Since the performance of parallel programs depends partly on the delay of the communication functions, it would be interesting to obtain the communication time. With this measurements it is possible to evaluate the blocking time of communication statements. This leads to the detection of communication bottlenecks and is very useful for program tuning and performance analysis.

4.3 Source Code Connection

A main idea in our approach to post-mortem debugging is to give the user the ability to change the source code immediatly after detecting an error. This is a common feature of traditional sequential debuggers and therefore had to be included in a parallel debugging approach as well.

The tracing of source code connections does not depend on the level of amount but is selectable as separate feature. It may be useful to have the source code connection with minimum level of amount as well as with maximum level.

The connection between the events and the corresponding line of code consists of the two items source file name and line number. During the inital program run the monitor stores these items for each event. With the source code connection available, ATEMPT automatically starts a filebrowser, retrieves the source file and marks the line of code, that contains the statement which is responsible for the event.

The underlying data structures presented in this section are not only used for monitoring but also for ATEMPT and our parallel simulator PARASIT.

5. Storing Tracefiles

5.1 Preparation Part/Storage Part

The most critical part of the monitoring overhead is the time consumption when storing the produced data. In our approach this time is mainly affected through the monitoring strategy, which divides the monitoring process into two parts. A first part prepares the data and therefore must be fully integrated in the computing process. The other one takes the prepared data, generates additional information and stores the eventdata for further processing.

The interesting question is, how much the computing process is delayed, while the monitor stores the gathered information. As there is no possibility that the first part -

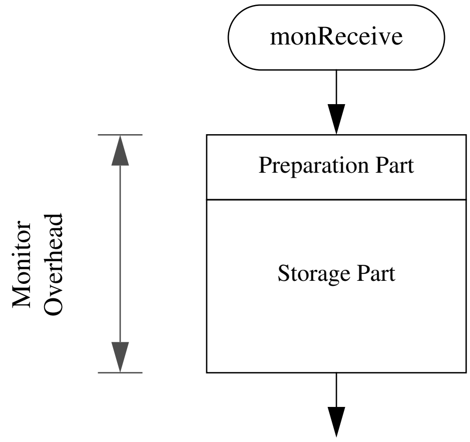

Figure 6a. Strategy Direct Overhead

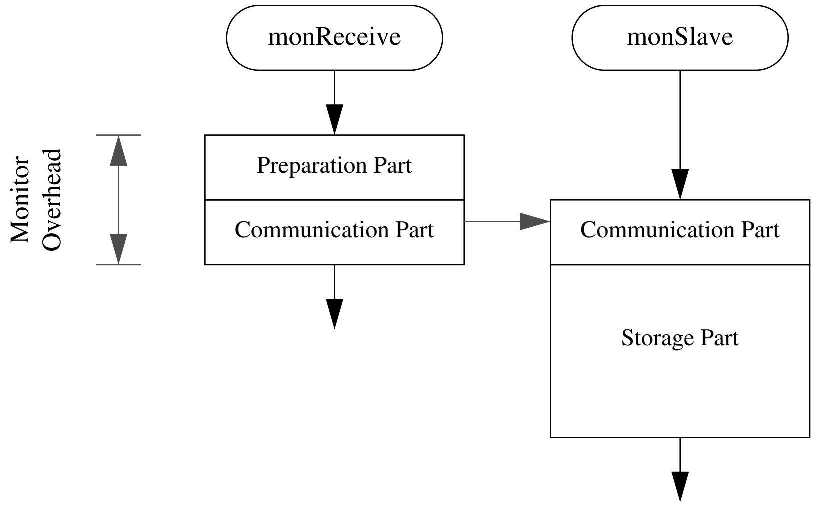

Figure 6b. Strategy Cube Overhead

the preparation part - is departed from the computing process, the only solution to this problem is to move the *storage part* from the computing process to a separate monitoring process. This can be achieved with our monitoring strategies.

The *direct* strategy includes all monitoring functionality directly on the computing node (figure 4a). This implies, that the *storage part* of the monitor also delays the monitored process and therefore generates a high monitoring overhead rate. On the other hand, the *monitoring cube* strategy includes only the necessary preparation part in the computing nodes (figure 4b). The *storage part* plus some additional data generation is done on dedicated monitoring slave nodes. In this model the time consuming data-

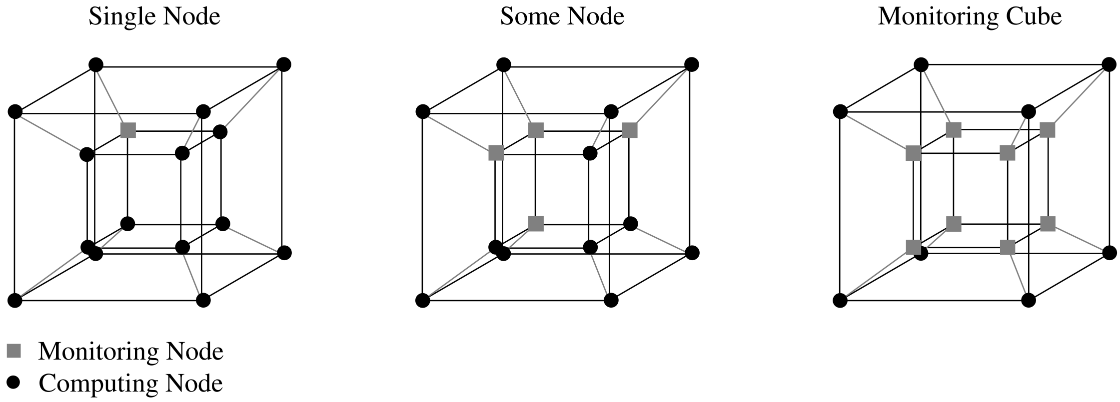

Figure 7. Monitoring Node Strategies

storage is accomplished on the monitoring slaves, which have to receive the prepared data by interprocessor communication. Therefore the monitored program has to include a communication part to transmit the eventdata.

5.2 Monitoring Strategies

With EMU the user has the possibility to select between the different strategies depending on the available hardware. To get more information about the different monitoring strategies, we will give a short description about them.

For the model *direct* the preparation part is the collection of the event attributes as specified through the amount level above. The *storage part* takes all information and performs the necessary I/O-statements for the appropriate tracefile access. The time for the diskaccess depends on several factors like I/O-hardware, operating system functions and program implementation. The most important factors are:

- I/O-Buffer size
- I/O-Flushing enabled/disabled

The I/O-Buffer size determines how many trace events are collected before actually writing them to the tracefile. A collection of monitoring events is only useful, whenever the traced program runs correctly and therefore no error monitoring is needed. If the debugger is used for the location of errors in the communication statements, it must be possible to write all information immediately to provide the user with all needed eventdata up to the state of termination. Another fact in tracefile-access is the network traffic, which depends on the actual running program, but this part strongly depends on the operating system and the hardware and needs some further investigations.

The *storage part* described above is used for the *direct* strategy as well as the *process* strategy, where a separate task on the computing nodes performs the monitoring activities. The advantage of the *process* strategy is obtained, if the computing process has a lot of idle time where the monitoring process on the same processor can use this time to perform the storage activities.

The other monitoring strategies - *single node*, *some nodes*, *monitoring cube* - are designed to split off the *storage part* from the preparation part. There dedicated monitoring slave nodes provide the functionality to perform the storage of the generated eventdata, while the computing nodes only have to prepare the data and transmit them to the appropriate slave node (see figure 7).

The model *single node* only provides one monitoring node for all computing slaves. The disadvantage of this model is that monitor communication raises network traffic, which means again intrusion on the running program. To reduce this influence, the use of more than one monitoring slave is a possible solution. This is achieved by using the strategy *some nodes*, where the *storage part* of the monitor is done on several monitoring slaves. The advantage of this model is that the communication links between neighbouring computing and monitoring nodes are only used for monitor communication and do not affect network traffic between computing nodes. The network traffic is only influenced by computing nodes, which do not have a monitoring node in their neighbourhood.

Improvements to this situation can be made by choosing the monitoring strategy *monitoring cube*. The disadvantage of the *monitoring cube* strategy is the loss of half the processing cube, but on the other hand each computing node has a dedicated monitoring node and a reserved link to this node for the monitor communication. Therefore the communication part is never delayed due to network traffic.

6. Implementation

Another differentiation about the monitoring overhead can be made concerning the level of implementation. As high programming language levels provide a high portability, they also consume more memory and time than low level assembler implementations. Our distinction between the levels of implementation is the following:

- function level
- macro level
- assembler level
- trap level

On function level, the traced statements are replaced with the monitoring statements within a parsing environment. A graphical user interface provides selectors for the source file. It is written in C++ with the Application Framework ET++ [Gamm91].

The monitor parsing environment follows the look and feel concept, which provides the user a simplified interface to the monitor. The user can select specific parts for his investigations and variables that should be inspected with a simple mouse click. After completing the user activities, the environment passes the changed source files to the compiler and produces a run-time version with the monitoring routines.

On macro level the substitution from the original communication functions to the monitoring functions is done through the C-language precompiler. The user has to insert a C-header file, which replaces the original statements with our own monitoring macros. After recompiling the sources all monitoring activity is selectable via environment-variables.

The assembler level provides a special communication library which replaces the original message passing interface at linktime. This will decrease the monitoring overhead because low level implementations are usually faster than high-level language programmes.

At the trap level the interrupt service routine is replaced whenever a communication statement is called at runtime. This means that the interrupt routines of each monitored function are changed to monitoring routines.

The facilities of function and macro level are mostly hardware independent or need only small adjustments to different hardware systems. Although the current implementation of the monitor is only available for the nCUBE 2, we designed this concept with regard to other architectures like the Parsytec MC-3 DE/1.

7. Conclusion

Debugging of parallel programs is one of the hard tasks of parallel software engineering. Due to the concurrently executing processes and communication established between these processes, the set of sequential programming errors is extended by race conditions, deadlocks and stampede effects. To overcome this problem we are developing a debugging toolset which helps the user in the testing and debugging phases of parallel program development.

The toolset consists of several modules for the different tasks in the debugging process. The first part is the monitoring tool EMU, which produces tracefiles from a running program. These tracefiles can be analysed using ATEMPT and the trace driven simulator PARASIT. With these tools it is possible to detect most errors in parallel programs. Through source code connection the changes in the source code can be made as soon as an error is detected.

This paper describes the event monitoring utility EMU, which performs monitoring of parallel programs. The main goal of EMU is to produce a minimal overhead in time and memory consumption. To achieve this goal several strategies were implemented and are presented in this report.

The next step in the development of these tools is the parallel simulator PARASIT. It is supposed to provide the functionality of trace driven simulation on the basis of the EMU-tracefiles.

Although the tools are in a prototype state yet, the whole toolset is highly accepted by the programmers at our campus. It has already proven to be a powerful tool for error detection. This gives us a lot of feedback for our future work.

Remark

An executable version of *ATEMPT* for SUN- and Linux-Workstations and some example tracefiles, as well as the *trcUtil* library for accessing EMU-tracefiles is available via World Wide Web from the following URL:

ftp://www.gup.uni-linz.ac.at/pub/debug

8. References

[ApMc88] W.F. Appelbe, Ch.E. McDowell, “Integrating Tools for Debugging and Developing Multitasking Programs”, Proc. of the SIGPLAN/SIGOPS Workshop on Parallel & Distributed Debugging, Madison, Wisconsin, pp. 78-88, May 1988

[ChMi91] J.-D. Choi, S.L. Min, "RACE FRONTIER: Reproducing Data Races in Parallel-Program Debugging", Proc. 3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP, Williamsburg, Virginia, pp. 145 - 154 (April 1991).

[ChSt91] J.-D. Choi, J. Stone, "Balancing Runtime and Replay Costs in a Trace-and-Replay System", Proc. ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, CA, pp. 26 - 35 (May 1991).

[Fidg 88] C.J. Fidge, “Partial orders for parallel debugging”, Proc. of Workshop on Parallel and Distributed Debugging, ACM, pp. 183 - 194 (1988).

[Gamm 91] E. Gamma, Objektorientierte Software-Entwicklung am Beispiel von ET++: Klassenbibliothek, Werkzeuge, Design, PhD. thesis, University Zurich, Swizerland (1991).

[Geis90] G. A. Geist, M.T. Heath, B.W. Peyton, P.H.Worley, “A Users’ Guide to PICL - A Portable Instrumented Communication Library”, ORNL/TM-11616, Oak Ridge National Lab (Aug. 1990)

[GrKr94] S. Grabner, D. Kranzlmüller, “Monitoring for Detecting Bugs and Blocking Communication”, Proc. of CONPAR 94-VAPP VI Proceedings, Linz, Austria (Sept. 1994)

[GrKr95] S. Grabner, D. Kranzlmüller, J. Volkert, “Debugging Parallel Programs using ATEMPT”, Proc. of HPCN Europe 95, Milan, Italy (May 1995)

[GrVo93] S. Grabner, J. Volkert, “Debugging Parallel Programs using Event Manipulation”, Proc. 1st Intl. Meeting on Vector and Parallel Processing, Porto, Portugal (Sept. 1993)

[HeFi93] M.T. Heath, J.E. Finger, “ParaGraph: A Tool for Visualizing Performance of Parallel Programs”, Technical Report Oak Ridge National Lab (Sept. 1993).

[Joyc 87] J. Joyce, G. Lomow, K. Slind, B. Unger, "Monitoring Distributed Systems", ACM Transactions on Computer Systems 5 (2), pp. 121 - 150 (May 1987).

[LeMe87] T.J. LeBlanc, J.M. Mellor-Crummey, “Debugging parallel programs with instant replay”, IEEE Trans. on Computing, pp. 471 - 482 (April 1987).

[nCUB92a] nCUBE Corporation, *nCUBE 2 Processor Manual Rel. 3.0*, 1992

[nCUB92b] nCUBE Corporation, *nCUBE 2 Programmer's Guide Rel. 3.0*, 1992

[McHe89] C. E. McDowell, D.P. Helmbold, “Debugging Concurrent Programs”, ACM Computing Surveys 21 (4), pp. 593-622, Dec. 1989

[MiCh88] B.P. Miller, J.-D. Choi, "A Mechanism for Efficient Debugging of Parallel Programs", Proc. SIGPLAN/SIGOPS Workshop on Parallel & Distributed Debugging, Madison, Wisconsin, pp. 141 - 150 (May 1988).

[RiTo92] M. van Rick, B. Tourancheau, “The Design of the General Parallel Monitoring System”, Programming Environments for Parallel Computing, IFIP, North-Holland, pp. 127-137, 1992

[SnHo88] D.F. Snelling, G.-R. Hoffmann, “A comparative study of libraries for parallel processing”, Proceedings of the Intl. Conference on Vector and Parallel Processors”, in Computational Science III, Parallel Computing, Vol. 8, (1-3), pp. 255-266, 1988