Monitoring Program Behaviour on SUPRENUM

Markus Siegle, Richard Hofmann,
Institut für Informatik VII, Universität Erlangen-Nürnberg,
Martensstraße 3, 8520 Erlangen, Germany,
e-mail siegle@immd7.informatik.uni-erlangen.de

Abstract

It is often very difficult for programmers of parallel computers to understand how their parallel programs behave at execution time, because there is not enough insight into the interactions between concurrent activities in the parallel machine. Programmers do not only wish to obtain statistical information that can be supplied by profiling, for example. They need to have detailed knowledge about the functional behaviour of their programs. Considering performance aspects, they need timing information as well. Monitoring is a technique well suited to obtain information about both functional behaviour and timing. Global time information is essential for determining the chronological order of events on different nodes of a multiprocessor or of a distributed system, and for determining the duration of time intervals between events from different nodes. A major problem on multiprocessors is the absence of a global clock with high resolution. This problem can be overcome if a monitor system capable of supplying globally valid time stamps is used.
In this paper, the behaviour and performance of a parallel program on the SUPRENUM multiprocessor is studied. The method used for gaining insight into the runtime behaviour of a parallel program is hybrid monitoring, a technique that combines advantages of both software monitoring and hardware monitoring. A novel interface makes it possible to measure program activities on SUPRENUM. The SUPRENUM system and the ZM4 hardware monitor are briefly described. The example program under study is a parallel ray tracer. We show that hybrid monitoring is an excellent method to provide programmers with valuable information for debugging and tuning of parallel programs.
Keywords: debugging, event-driven monitoring, multiprocessor, parallel program, performance evaluation, ray tracing, SUPRENUM, tuning.

1 Introduction

It is often very difficult for programmers of parallel computers to understand how their parallel programs behave at execution time, because there is not enough insight into the interactions between concurrent activities in the parallel machine. Programmers need to have detailed knowledge of the functional behaviour of their programs, and for the consideration of performance aspects they need timing information as well. Usually, methods such as profiling and accounting do not provide sufficient information, they only give summary statistical results. Therefore users often resort to rudimentary methods, such as writing log-files during program execution, in order to obtain debug information and performance information about their programs. But only a relatively small fraction of the needed information can be obtained that way. A major problem with multiprocessors is the absence of a global clock with high resolution. Global timing information is essential for determining the chronological order of events on different nodes of a multiprocessor or of a distributed system, and for determining the duration of time intervals between events from different nodes.

Facing this problem, our approach is to apply event-driven monitoring techniques [3] [8] [9] in order to find out how a parallel program behaves. In particular, we decided to use hybrid monitoring, which combines advantages of both hardware monitoring and software monitoring. Using software monitoring, it is relatively easy to relate the event traces obtained from the measurements to the measured program. But since monitoring is done within the object system (i.e. within the system under study), and therefore constitutes an extra workload, software monitoring changes the behaviour of the object system. Also, it is usually impossible to obtain global timing information because most parallel systems do not provide a global clock with high resolution. With hardware monitoring there is no intrusion and the timing problem can be solved by providing an external clock. But there is no easy way to relate the recorded signals to the source code of the measured program.

In hybrid monitoring, as in software monitoring, the program under study is instrumented by inserting additional instructions at points of interest. The execution of such a measurement instruction marks an event. It causes the output of measurement data, containing a token identifying the event and possibly some additional parameters, to an external interface. A hardware monitor is connected to the interface. It records the event stream coming from the interface, and stores the sequence of events together with the respective time stamps as an event trace. Since most of the work is done by the external hardware monitor, hybrid monitoring provides the capabilities of software monitoring at a much lower level of intrusion. The hardware monitor we use is a scalable distributed monitor system called ZM4. It is capable of simultaneously recording event streams coming from an arbitrary number of nodes. Since the ZM4 has a global clock with high resolution, events coming from different nodes can be chronologically ordered by their time stamps.

In this paper, we use hybrid monitoring techniques to study the behaviour of a parallel program on the German supercomputer SUPRENUM. The SUPRENUM project was launched in 1984 as a government-funded project, and implementation of a prototype started in 1986. The project resulted in a commercial product in 1990. SUPRENUM is a distributed-memory multiprocessor in which the processing nodes are interconnected by a hierarchical bus system. The SUPRENUM architecture and its software environment are discussed in more detail in Section 2.

In section 3 the essential features of the hardware monitor ZM4 are presented. Section 4 presents the application of our method to the implementation of a parallel ray tracing program on SUPRENUM. We show how measurements supported the implementation of a parallel program by providing programmers with valuable information about the behaviour of their program. Both debugging and tuning of the parallel program were supported considerably by the measurements.

2 The SUPRENUM Multiprocessor

2.1 Hardware Architecture

The SUPRENUM system [5] [11] [13] [2] is a MIMD-type multiprocessor consisting of up to 256 processors (nodes). It is a distributed-memory machine with a two-level interconnection network. All the elements comprising one processing node are accommodated on a single printed circuit board.

The main components of each node are a 32-Bit microprocessor (MC68020) operating at a clock rate of 20 MHz, 8 MByte of main memory protected by 2-Bit error-detection and 1-Bit error-correction logic, and four coprocessors:

- The paged memory management unit (PMMU) (MC68851) checks access rights and page violation when the node memory is being accessed by the CPU or at the beginning of DMA.
- The floating-point unit (FPU) (MC68882) executes scalar floating-point arithmetic.
- The vector floating-point unit (VFPU) consists of the Weitek chip set WTL2264/2265 and 64 KByte of fast static memory (vector cache). Peak performance is 10 MFlops for single-operation double-precision floating-point computations, and 20 MFlops in the case of chained operations. Peak performance is achieved even if one of the two operands is being read from main memory by DMA, provided a constant increment is used.
- The communication unit (CU) is a microprogrammable coprocessor which takes care of the data transfer between a node's main memory and other nodes in the system. The CPU initiates the communication. The communication unit then handles the entire data transfer including bus request, transfer with protocol checks, and bus release. The functions of the communication unit are realized mainly by gate arrays and hybrid modules.

Up to 16 processing nodes form a cluster. Nodes of the same cluster communicate via the cluster bus. In order to provide some degree of fault-tolerance, the cluster bus consists of two independent parallel buses, each having a transfer rate of 160 MByte/s. Thus the total bandwidth available for intra-cluster communication is 320 MByte/s.

Figure 1 (left) shows the components of one SUPRENUM cluster. In addition to the processing nodes, each cluster contains 3 or 4 special purpose nodes: there are up to 2 communication nodes which handle the communication between clusters. If a processing node in one cluster wants to communicate with another processing node in a different cluster, communication is done via a communication node. There is one disk controller node which can connect up to 4 disks to the cluster. Finally, there is one cluster diagnosis node which monitors the clusterbus and maintains statistical records. Only communication activities can be monitored by the diagnosis node.

The clusters are interconnected in a toroid structure by bit-serial buses, called SUPRENUM bus. Figure 1 (right) shows the 16 cluster SUPRENUM system and a front-end computer. A token ring protocol is employed for the SUPRENUM bus with a data transfer rate of 25 MByte/s. By duplicating the torus structure the bandwidth doubles and fault-tolerance is achieved because the clusters in a ring can always be reached via alternative routes.

2.2 The Programming Model

The SUPRENUM computer is a multi-user machine. Users can access the SUPRENUM kernel via a front-end computer. In order to execute a parallel program, a user must first request a certain number of clusters or nodes. If the requested number of resources is not available at the moment, the user has to wait. The code of the user program is then downloaded from the front-end computer to the partition assigned to the user. When the user's job terminates, all processors are released. There is a certain time limit which can be set by the operator, after which the resources assigned to a user are released, even if that user's job is not yet completed. This is done to prevent monopolization.

The following programming model is employed for SUPRENUM: user programs consist of one or more independent processes. A process can create other processes at any point of time. A user application starts with an initial process. Termination of the initial process causes termination of the whole application program. Otherwise a process can only be terminated by itself.

Processes communicate by sending and receiving messages. According to the specifications, both synchronous and asynchronous communication between processes are supported. Using synchronous communication, the sender of a message is blocked until the receiver of the message accepts the message. The sender cannot do any work while waiting for his communication partner to receive the message. In order to avoid this waiting time, asynchronous communication can be used. In this case the sender does not send the message directly to the receiver but to a mailbox associated with the receiver. The

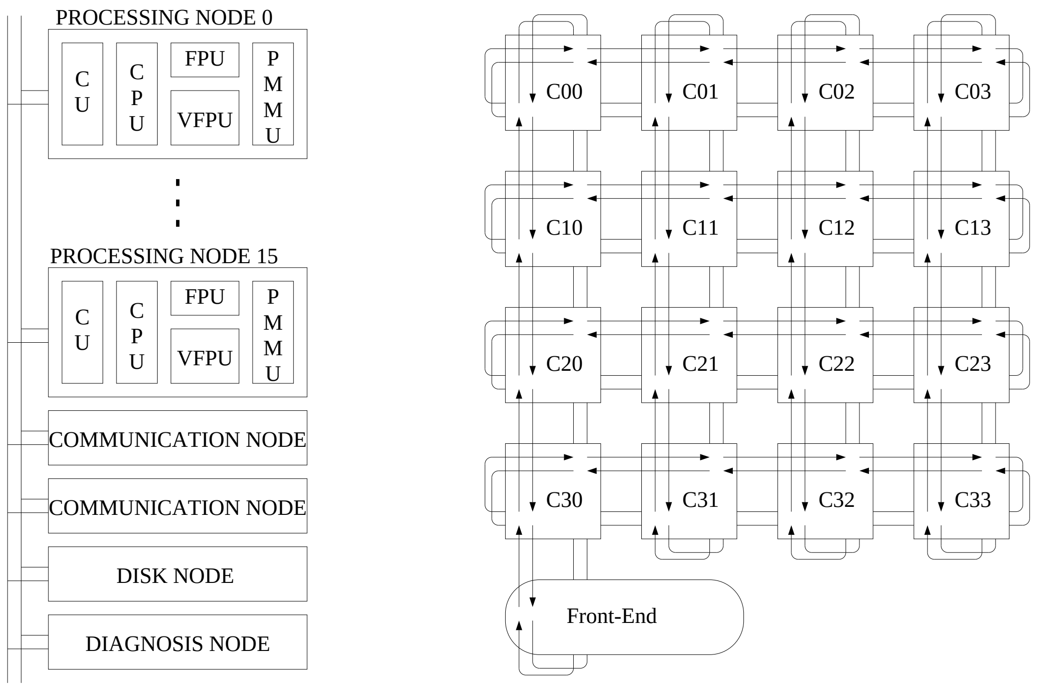


Figure 1: The SUPRENUM cluster (left) and the 16-cluster SUPRENUM system

sender can continue with his work once the message is placed in the receiver's mailbox. The message is actually read from the mailbox by the receiver at a later time. In Section 4 we will discuss the behaviour of SUPRENUM's synchronous and asynchronous communication in detail.

3 The Hardware Monitor ZM4

3.1 The ZM4 Architecture

The ZM4 is a hardware monitoring system which has been developed and built at the University of Erlangen-Nürnberg [4] (The name indicates that the ZM4 is the fourth generation of monitor systems built at our University). It is designed to measure arbitrary parallel and distributed systems. Consequently the ZM4 itself is a distributed system which is scalable and adaptable to any object system. This universality is an important feature of the ZM4, considering that most sophisticated monitoring systems are dedicated to one special object system [1] [3] [7] [14].

We now describe the components of the ZM4 in a bottom-up fashion. The central component of the ZM4 is the dedicated probe unit (DPU) which consists of probes interfacing to the object system, an event detector, and an event recorder. The event detector detects relevant measurement information coming from the object system interface. The probes and the event detector are the only parts of the ZM4 that depend on the object system.

The event recorder of the ZM4 is realized as a plug-in board for a monitor agent (MA). Standard PC/AT computers are used as monitor agents. One event recorder can record up to four independent event streams. Upon a request signal the event recorder inputs data coming from the event detector. It stores this data together with a time stamp and a flag field into a FIFO buffer of size $32K * 96$ bits. The contents of the FIFO buffer are written simultaneously onto the disk of the monitor agent. The FIFO is needed as a high-speed buffer to ensure that no events get lost during bursts of events.

The clock of the event recorder has a resolution of 100 ns. About 10000 events per second can be written from the FIFO buffer onto the disk of the monitor agent. This limit is due to the disk transfer rate of the monitor agent. However, a bandwidth of 120 MByte/s at the input of the FIFO allows for peak event rates of 10 millions of events per second during bursts of events.

In order to monitor larger object systems, two or more DPU's have to be employed. In this case the local clocks of the event recorders have to be synchronized to obtain globally valid time stamps. Another plug-in board, called measure tick generator (MTG), is used for that purpose. It constitutes the master part of the global clock of the ZM4. It is connected to the event recorders via the tick channel. The local clocks of the event recorders can be started simultaneously by a signal on the tick channel. A manchester-coded signal which is transmitted continuously via the tick channel prevents skewing of the local clocks. Thus the local clocks can provide globally valid timing information. For details about the global clock of the ZM4 see [4].

Up to four DPU's can be plugged into one monitor agent. If one wishes to use more than four DPU's, more than one monitor agent is needed. It is important to note that there is still only one measure tick generator connected to all event recorders by the tick channel. All monitor agents are connected to a control and evaluation computer (CEC) by the

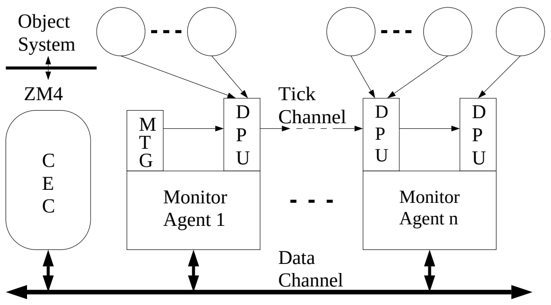


Figure 2: The ZM4 distributed, scalable hardware monitor

data channel (an Ethernet using TCP/IP). Figure 2 shows the general configuration of the ZM4.

When a measurement has been carried out, the event traces recorded by the event recorders and stored on the disks of the monitor agents are transmitted via the data channel to the control and evaluation computer. There the local traces can be merged to one global trace, since events can be sorted according to their globally valid time stamps. Evaluation of the global trace is done using the SIMPLE software package [10] [4] which provides tools for statistical analysis, visualization, and animation of measurement data.

3.2 Adaptation of ZM4 to SUPRENUM

As mentioned in the previous section, the interface between object system and event recorder depends on the object system. In the past, the ZM4 has been used for monitoring a great variety of object systems such as Transputers, high-speed networks, clusters of workstations, and robot systems. Each such object system may provide a different hardware interface to which measurement data is output.

In the case of SUPRENUM we considered two candidate interfaces for outputting measurement data. Each processing node has a serial terminal interface (V.24) and a seven segment display on its front cover. The terminal interface is intended to be used by service personnel. The seven segment display displays the internal state of communication firmware. Data transfer via the terminal interface is slow (less than 20 KBit/s). It would take more than 2.4 ms to output 48 bits of event data, not including time for context switching. Therefore we decided not to use the terminal interface.

The seven segment display is driven from a gate array on the processing node board. It can display only 16 different patterns. Under normal operating conditions, the seven segment display is not intended to be addressed by the user. We wished to output event data which is 48 bits wide via the seven segment display. To code the event, 16 bits of the event data are used, and a parameter field of 32 bits is provided for outputting additional information relevant at the point of the program where the event is initiated. In order to output 48 bits of data via a display which can only display 16 different patterns, we devised the following scheme: one pattern is used as a triggerword T which signals to the monitoring hardware that measurement data will follow. The 48 bits are output as a sequence of 16 pairs $T\ m_i$ like

$$T\ m_0\ \ T\ m_1\ \ \ldots\ \ T\ m_{15}$$

where each m_i is a pattern that encodes 3 bits of the original 48 bits. There are two essential conditions:

- The triggerword T must be reserved for this application, since it signals to the event detector that measurement data will follow.
- The output of a pair $T\ m_i$ must be an atomic action. No other information must be output to the seven segment display between T and m_i.

The routine that can be called from the user program in order to output data via the seven segment display was implemented in cooperation with the SUPRENUM company. It is called as

$$hybrid\_mon(p1, p2)$$

where $p1$ is a 16-Bit integer defining the event and $p2$ is a 32-Bit parameter which can be used to output additional relevant data. One call of the routine *hybrid_mon* takes less than one twentieth of the time that would be needed to output an event via the terminal interface. This results in a very low level of intrusion and ensures that the time needed to output an event is more than two orders of magnitude smaller than the duration of the measured activities.

An interface was built whose probes are plugged into the socket of the seven segment display on one side and which connects to the event recorder of the ZM4 on the other side as shown in Figure 3. Its event detector con-

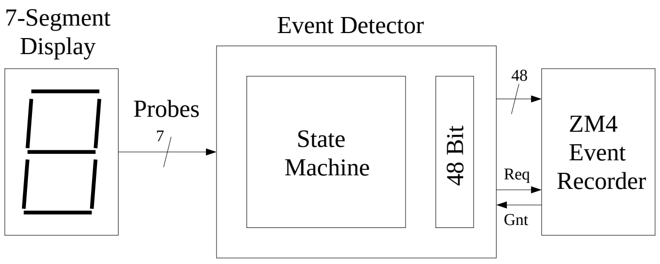


Figure 3: The interface between SUPRENUM and ZM4

tains recognition logic for the triggerword T and reconstructs the original 48 bits of the event data from the sequence $T\ m_0\ \ T\ m_1\ \ \ldots\ \ T\ m_{15}$. It is realized as a state machine in programmable logic. Once a 48-Bit event is assembled the interface issues a request signal and the event is recorded by the event recorder of the ZM4.

4 Performance Evaluation of a Parallel Ray Tracer

As mentioned in the introduction, the main difficulty when writing parallel programs is the lack of insight into the internal activities of the parallel program and into the mutual effect of the user program and the system software of the underlying parallel architecture. Programmers wish to know more about the functional behaviour as well as the performance of their parallel programs. These problems are also

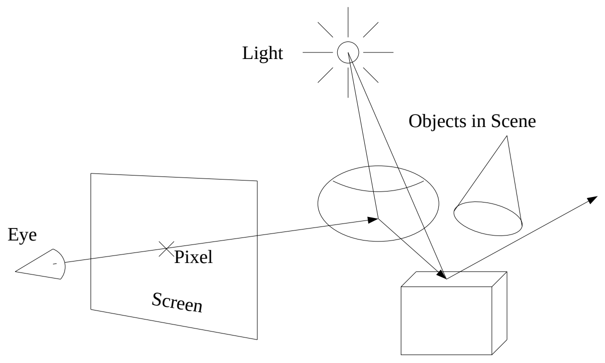


Figure 4: The basic concept of ray tracing

encountered by people writing programs for the SUPRENUM multiprocessor. In this paper, we investigate how hybrid monitoring techniques can support the implementation of parallel programs for a novel machine. Both debugging and tuning are addressed.

We have chosen ray tracing as an attractive example application, and we have implemented a parallel ray tracing program for SUPRENUM [12]. It is important to note that it was not the major aim of our work to build a highly sophisticated ray tracer, but to develop methods which make writing of correct and efficient parallel code easier.

4.1 Ray Tracing

An important field of research in computer graphics is the generation of realistic images from formal descriptions of a scene. Natural phenomena such as light and shadow, reflection and refraction, or depth of field have to be simulated in order to get such high-quality images. Ray tracing [15] [6] is one of the most sophisticated and most powerful methods for image generation.

The basic concept of ray tracing is illustrated in Figure 4. The idea is to follow the trajectory of a ray (the eye ray) originating at the eye of the observer and going through one particular pixel of the image plane into the scene. The colour value of the eye ray (i.e. of its associated pixel) has to be computed. To do this, the closest point of intersection between the ray and objects in the scene is determined. The colour value of the pixel depends on the surface properties of the object at the intersection point in various ways. First of all, the colour value depends on the colour of the object at the intersection point and how that point is being illuminated by light sources. If the object is "shiny", i.e. if light hitting the object is reflected, the colour value also depends on the colour of a reflected ray originating at the intersection point. Finally, if the object is not opaque, the colour value of the pixel also depends on the colour of a transmitted ray originating at the point of intersection. The colours of both the reflected ray and the transmitted ray are computed recursively. The colour of the eye ray is a combination of the colour of the object, the colour of the reflected ray, and the colour of the transmitted ray.

Ray tracing is a very time-consuming process. Note that 256K pixels have to be processed in order to compute a 512 by 512 image. Depending on the resolution of the image to be created, the complexity of the scene, and the shading model used, it can take between seconds and days to render a scene. This is a great challenge for computer architects and programmers. In the past many acceleration techniques for ray tracing have been suggested [6].

Parallelization is a very promising approach for improving the performance of a ray tracing program. By dividing the work to be done between a large number of processors, computation time can be reduced significantly. Many different ways to parallelize ray tracing have been proposed and tested. There are two basic schemes to distribute the computation between several processors:

- Using *object partitioning,* each processor takes care of a certain fraction of the objects in the scene to be rendered.
- With *ray partitioning,* different rays are processed by different processors.

The second scheme, *ray partitioning,* works as follows: each processor computes a part of the final image by computing the colour values of a subset of all rays which have to be processed. In doing this, each processor can work completely independently of all other processors involved. There is no communication between processors until the end, when the complete image is put together from the local results. For processing a ray, each processor needs information about the whole scene to be rendered. This means that the information contained in the scene description must be replicated on each processor, which requires redundant storage. This may be a disadvantage because scene descriptions are often very long and need a lot of memory.

With ray partitioning, it may either be predetermined which rays are processed by a particular processor (static ray partitioning), or the assignment of a ray to a processor may be made during the computation of an image (dynamic ray partitioning). The performance of static ray partitioning is often quite poor because the computation time for a single ray varies significantly, and therefore it is difficult to predict how long it will take to process a certain subset of rays. This results in a load balancing problem which can be at least partly solved by assigning discontinuous subsets of rays to the processors, instead of assigning continuous subsets such as rectangular patches to the processors.

4.2 Parallelizing Ray Tracing for SUPRENUM

For our parallelization of ray tracing we chose a dynamic ray partitioning approach in which the ray tracing algorithm is executed in a distributed manner by one master processor and several servant processors. As shown in Figure 5, the master communicates with all the servant processors, but there is no communication between any two servant processors. The master is responsible for control of the high-level flow of the algorithm, whereas the actual tracing of rays (the geometric intersection operations, mapping transformations, etc.) is executed by the servants.

Figure 6 shows the basic structure of the master and servant processes. The horizontal bars in the figure denote instrumentation points. At these points, measurement instructions were inserted into the program. The master adminis-

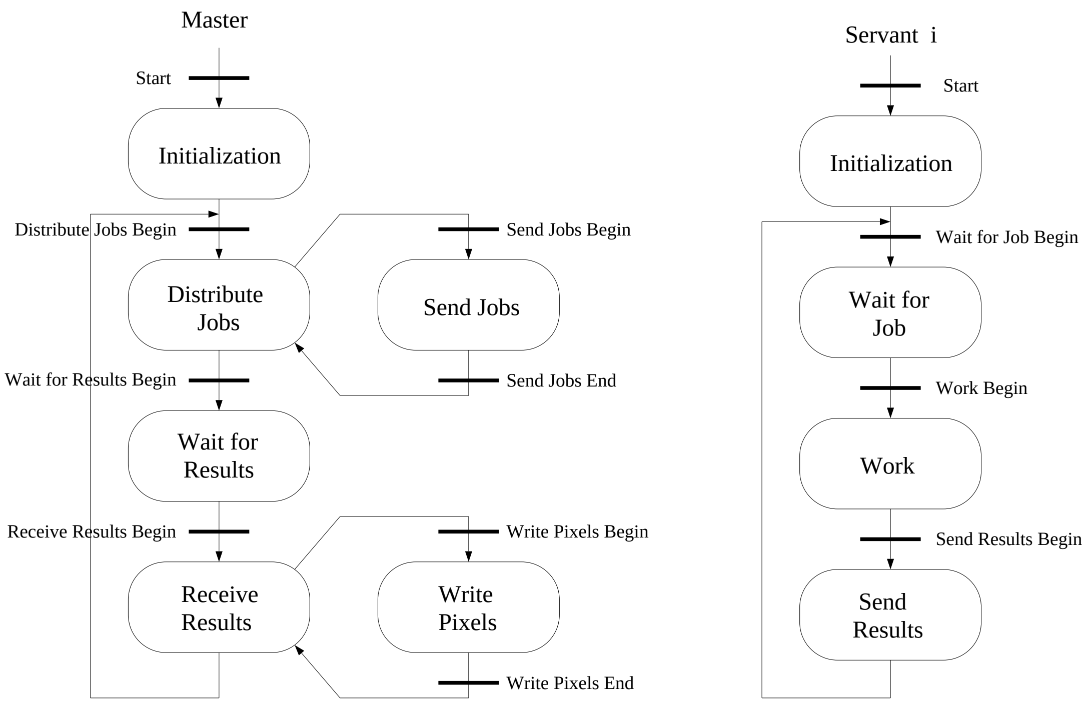


Figure 6: Basic structure of master and servant processes

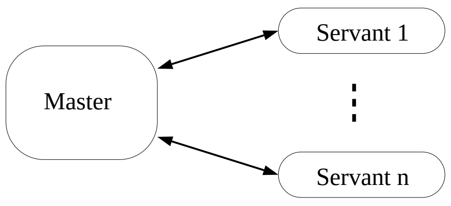


Figure 5: Process structure of the ray tracer

trates the work to be done. He always keeps a certain number of unfinished pixels in a queue. While there are more pixels to process, the master assigns jobs to the servants ("Distribute Jobs", "Send Jobs"), collects the results returned from the servants ("Receive Results"), and writes the output picture file ("Write Pixels"). The number of times the code for "Send Jobs" and "Write Pixels" is executed in each loop may vary. An oversampling scheme, in which more than one ray is computed per pixel in order to reduce aliasing problems, is also organized by the master. The jobs assigned to the servants consist of bundles of one or more rays whose colour values have to be computed.

The servants receive messages containing a job, trace the rays belonging to a job ("Work"), and return the results to the master ("Send Results"). They can work independently of each other because they all have the complete scene information available. Each servant has to communicate with the master to receive jobs and return results, but there is no communication between the servants.

The maximum number of outstanding jobs assigned by the master to one particular servant is limited by a window flow control scheme which works as follows: initially the master has a fixed number of credits from each servant. The master may send jobs to a servant as long as there are credits from that servant available. With each result the master gets one credit back from a servant. This load balancing scheme prevents flooding of the servants with jobs coming from the master, but it also ensures that the servants always have enough work to do to keep them busy.

The time to compute a ray varies considerably. For instance, a ray which does not intersect any object of the scene gets assigned the background colour of the picture without any further processing. A different ray may hit an object, generate secondary rays which in turn may generate more rays recursively, so that much more processing is required. The window flow control scheme described above guarantees that the activities of the servants are completely decoupled. Furthermore, the assignment of rays to processors is completely dynamic. Therefore processing of "long" rays on one servant does not cause other servants to wait.

It is easy to see that the master constitutes a hot-spot for communication because he must communicate with all the servants. We expected that this would not cause any problems, provided the following conditions hold:

- Communication is sufficiently fast.
- The time needed to process a ray is significantly longer than time needed to do administrative work associated with a ray.

4.3 Evolution of the Parallel Program

In this section we describe four versions of a parallel ray tracer. Evaluating measurements of each version motivated the changes introduced for the following version.

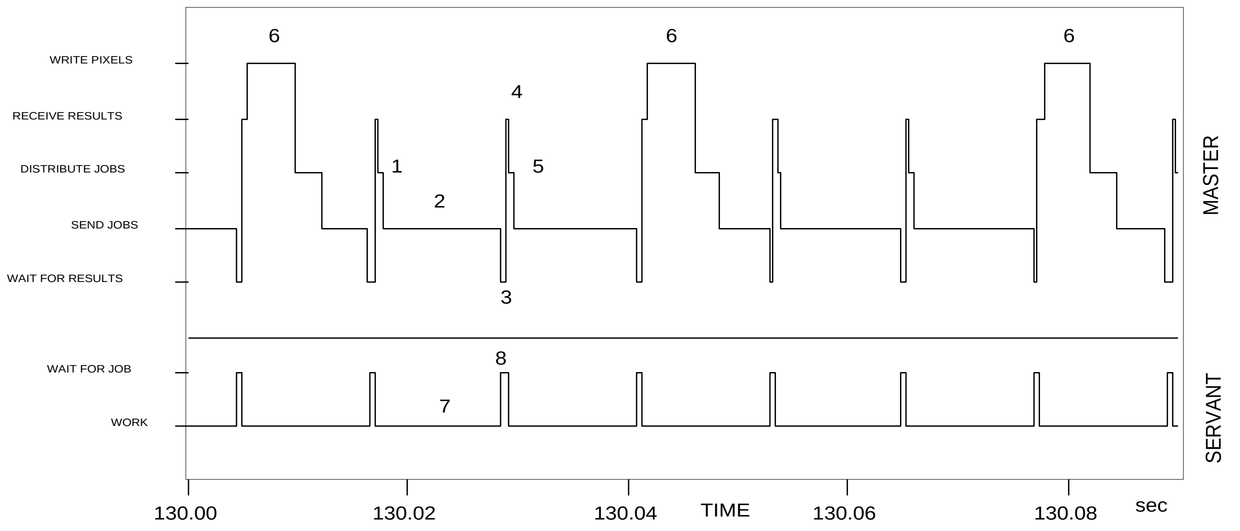


Figure 7: Behaviour of the mailbox communication (ray tracer on two processors)

Version 1: SUPRENUM's Mailbox Mechanism
We started by implementing a basic ray tracing program for SUPRENUM. The implementation of the first version was led by the following considerations:

It is important for the communication between the master and the servants that the sender of a message does not get blocked until the message is actually received by the receiver. Otherwise a lot of time would be spent idling while waiting for the communication partner. Therefore asynchronous communication has to be employed. In particular, we used the mailbox mechanism of the SUPRENUM system. A mailbox is a light-weight process owned by the receiving process. The sender of a message does not send the message directly to the receiver but to the receiver's mailbox. The receiver reads his mailbox whenever he wishes to do so. According to the specifications of SUPRENUM's mailbox mechanism the mailbox process is always in a receive state and therefore the sender of a message will never be blocked.

However, measurements of the mailbox communication between the master and one of the servants reveiled a different behaviour. Since the mailbox is a (light-weight) process, it must be actually running in order to receive a message. The mailbox process is running on the same processor as the receiving process (together with several other light-weight processes) in a time-sharing manner. The scheduling strategy used is plain round-robin. However, instead of using time-slicing, each process that is scheduled may either run until it gets blocked or until it decides to relinquish the processor deliberately. The sender of a message is blocked until the mailbox process on the receiver's processor is actually scheduled. This may not be the case until the receiver himself becomes blocked because he is waiting for a message or because he wants to send some message himself. Consequently, (asynchronous) mailbox communication behaves very much like synchronous communication. This can be observed in the Gantt-chart shown in Figure 7.

A Gantt-chart is a time-state diagram which depicts program activities during the measurement. For the diagram in Figure 7 we measured our ray tracing program running on two processors. The activities of the master processor and the servant processor are shown over a common time axis. Communication between the master and the servant is done using mailboxes. One can observe in the Gantt-chart that the master goes through the following stages (numbers refer to locations in the chart):

- During the activity "Distribute Jobs" (1) the master refills the pixel-queue and does some more administrative work.
- During "Send Jobs" (2) he sends a job (in this case consisting of a single ray) to the servant. After the send activity the activity "Wait for Results" (3) begins.
- In "Receive Results" (4) the master receives a message coming from the servant. The message contains results computed by the servant. However, these are not the results for the job just sent, but for a previous job (remember with the window flow control scheme the number of outstanding jobs per servant may be greater than one).
- "Receive Results" is followed by the next "Distribute Jobs" (5) activity.

Some of the master's cycles also contain a write activity (6) (in the window shown in Figure 7 this is the case in every third cycle). The duration of "Distribute Jobs" is significantly longer after such a write activity because new pixels must be inserted into the pixel-queue, after pixels whose computation is completed have been written onto disk.

The servant works on a job ("Work", 7), returns the results, and waits for the next job ("Wait for Job", 8). It can be seen that the servant spends most of the time in the "Work" state, so servant utilization in this measurement is very good. Since there is only one servant present, the master can easily keep him busy and the servant has always enough work to do.

It can be observed in the diagram in Figure 7 that the master becomes blocked during the "Send Jobs" activitiy. The transition from "Send Jobs" to "Wait for Results" ($2 \rightarrow 3$) on the master processor can only occur in a synchronized manner with the transition from "Work" to "Wait for Job" (7

$\rightarrow$ 8) on the servant processor. This is a very disappointing result, because in using mailbox communication we expected that the master's "Send Jobs" and the servant's "Wait for Job" would be decoupled. The reason for the observed behaviour is as follows. The servant's mailbox process can only be scheduled after the servant relinquishes the processor because he is waiting for a message. Only then can the master place his message in the servant's mailbox. This results in a synchronous behaviour of SUPRENUM's mailbox communication.

We monitored another run of this version of our ray tracer. The behaviour of the program is illustrated in the Gantt-chart shown in Figure 8. These results were obtained from monitoring the program when it was rendering a scene of moderate complexity (the scene contained 25 primitive objects). The program ran on 16 processors, i.e. there is one master and 15 servants. Only one servant is shown in the figure but the other servants behave similarly. It can be easily seen that the program's performance is very poor. The servants are only working about 15 % of the total time. Each servant spends the rest of the time waiting for the master to accept the results of the current job and to give the servant a new job. In this measurement the window size for the number of outstanding jobs per servant was 3, and a job consisted of one single ray.

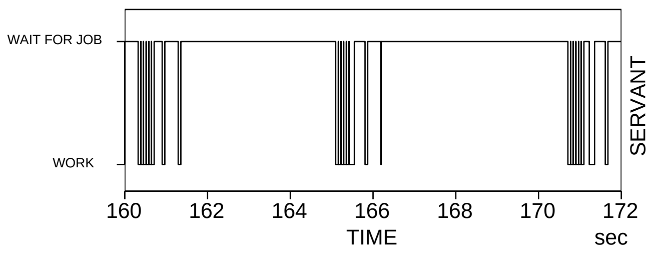


Figure 8: Servant utilization using mailbox communication(ray tracer on 16 processors)

Version 2: Communication Agents (one direction)
Having evaluated this and other measurements, we decided to implement our own asynchronous communication in order to avoid such long waiting times. For the communication from the master to the servants we introduced a pool of light-weight processes which we call communication agents. Their task is to forward a message from the master to one of the servants. The agents are running on the same processor as the master. Whenever the master wishes to send a message to a servant he indicates this fact to an agent, who is currently not engaged in some other communication, by setting a shared variable. This agent will forward the master's message to the servant. If no free agent is available a new agent is created and added to the pool. Measurements showed that the number of agents created remains quite small. After the indication the master relinquishes the processor and all agents will be scheduled. Context-switching between light-weight processes belonging to the same team of processes is cheap (less than 1 ms).

Ray tracing the same scene as above with the modified program, servant processor utilization improved to about 29 %, as can be seen in Figure 9 (during the measurement the servants spent about 29 % of the total time in the "Work" state). A pool of 5 communication agents was created for rendering this scene on 16 processors. For clarity of the figure only one communication agent is shown. Also, as before, only one servant's activities are shown in the figure.

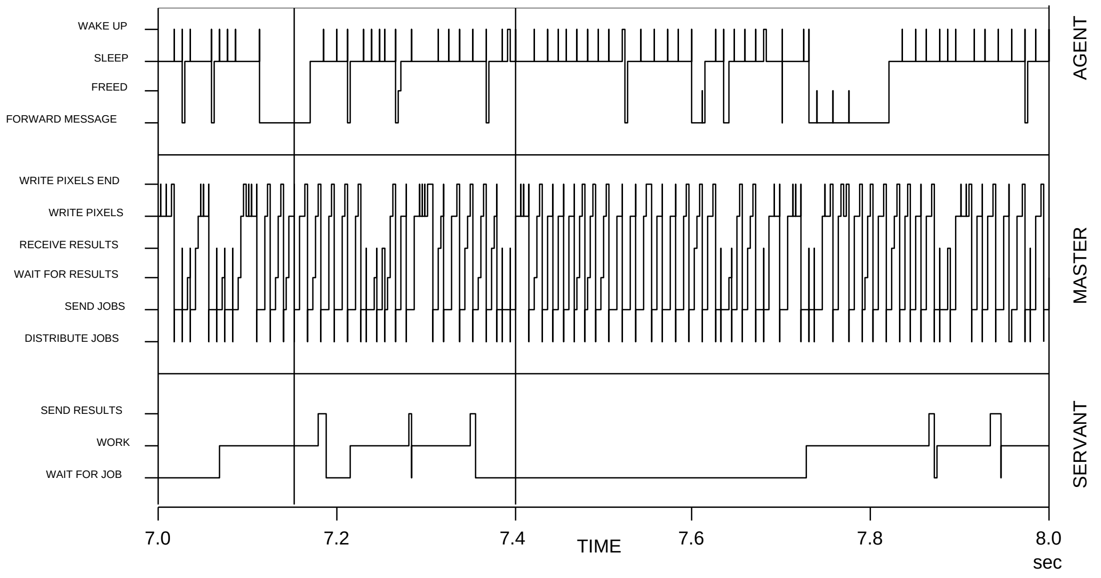
Figure 9 (bottom) gives a more detailed view on the activities going on (numbers refer to locations in the chart): as in Figure 7, one can observe the cycles the master goes through. Again, writing the output file is not done in every cycle. In one of the cycles there are several pixels written to the output file at a time (1). This is because pixels have to be written in correct ordering. So, whenever a continuous stretch of pixels has been processed, the results are written onto disk. Waiting times of the master are not significant. Therefore one way to solve the still existing communication bottleneck at the master would be to reduce the number of messages sent, by sending jobs consisting of more than one ray, as described below.

For the Gantt-charts shown in Figure 7 and Figure 8 the beginning of the servants' "Send Results" activity had not been instrumented. In these charts there is a direct transition from "Work" to "Wait for Job". Since we wished to know the duration of the "Send Results" activity, we inserted an additional measurement instruction at the beginning of "Send Results" for the charts in Figure 9. In Figure 9 we can see that the servant goes through the stages "Work" (2), "Send Results" (3), and "Wait for Job" (4).

The behaviour of the communication agents can be observed in the Gantt-chart in Figure 9 (bottom): if an agent is scheduled ("Wake Up", 5) and finds that there is no message to be forwarded, he goes back to sleep immediately ("Sleep", 6). Otherwise he takes the message, forwards it to the receiver ("Forward", 7), is freed whenever the message is received by the receiver ("Freed", 8), and goes back to sleep ("Sleep", 9). It can be seen from the chart that the time an agent spends in the "Freed" state is extremely short (8).

Version 3: Communication Agents (both directions)
Using communication agents improved the servant processor utilization by almost 100 %, but a utilization of 29 % is still very poor. As introducing communication agents for sending messages from the master to the servants turned out to be helpful, we decided to use agents for the reverse communication (from a servant to the master) as well. Apart from this, we wanted to relieve the communication problem by reducing the total communication volume. Sending a message for every single ray is certainly not the best strategy. Therefore we reduced the number of messages to be sent by sending bundles of rays instead of single rays. More specifically, 50 rays are now sent to a servant at a time, and the servant returns 50 results in one message. Measurements showed that these changes resulted in a further improvement of the ray tracer's performance. Now the servant utilization was more than 46 %.

Version 4: Further Tuning
Measurement results from the previous version indicated that the communication hot-spot at the master is still the major problem. We wished to know whether it would be advantageous to further increase the bundle size and chose a bundle



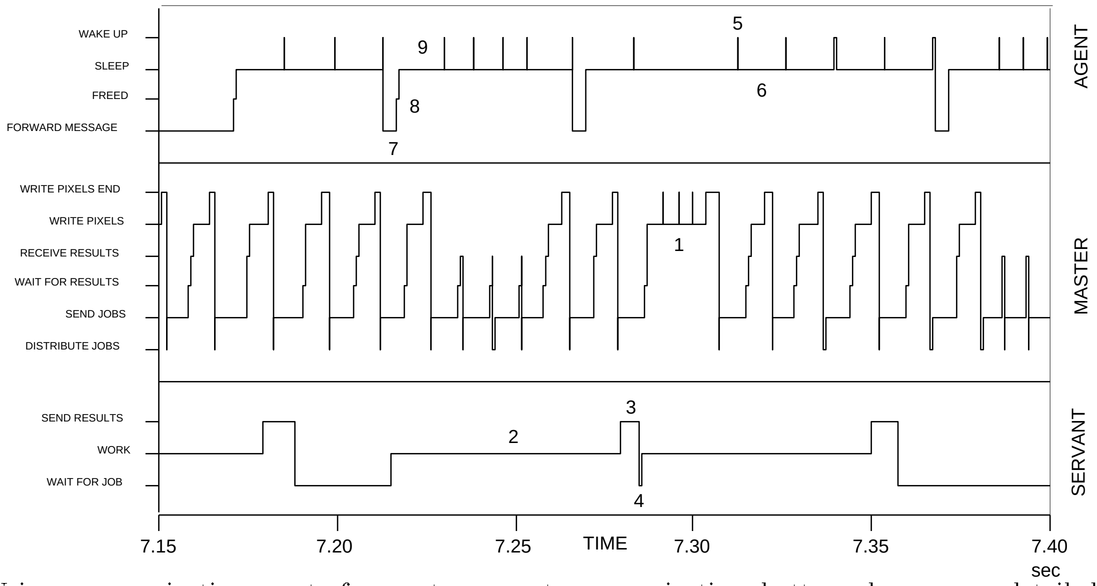


Figure 9: Using communication agents for master-servant communication, bottom shows more detailed view of measurement

size of 100 for the next measurement. A bundle size of 100 is still small compared to the total number of rays which have to be computed for a whole image (remember for a 512 by 512 image 256K rays have to be processed even if no oversampling is done). Therefore distributing jobs of size 100 to the servants does not cause any significant load balancing problems. Also, a minor programming error in the previous version of the program had been detected, namely the choice of an inadequate constant for the length of the master's queue of pixels to be computed. This lead to a situation in which there were not enough pixels in the pixel-queue to constitute a sufficient amount of work for the servants. With these changes our ray tracer finally achieved 60 % servant processor utilization when ray tracing the example scene. The improvement of servant utilization for the example scene using the four described versions of our program is shown in figure 10. For this scene, the initial servant utilization of about 15 % could be improved to 60 %.

Rendering Complex Scenes

The more complex a scene, the more time it takes to trace a single ray. More complex scenes result in a workload with relatively more computation and less communication, i.e. a good servant processor utilization can be achieved more easily when rendering complex scenes. As mentioned before, the example scene rendered during the measurements discussed above was only of moderate complexity. Therefore, this scene was a hard test candidate for our ray tracer. Rendering a more complex scene comprising more than 250 primitives (a fractal pyramid) we found that the servant processors reached a utilization of over 99 %. Due to the complexity of this scene the master did not become a bottleneck although he had to

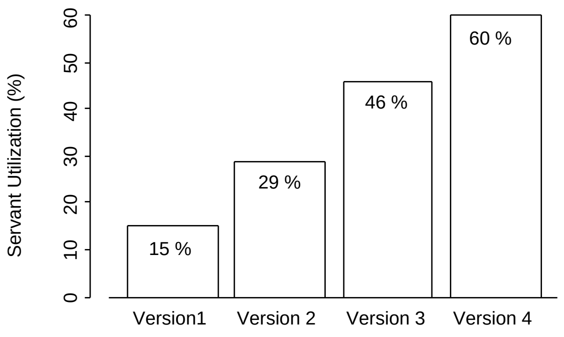


Figure 10: Improvement of servant utilization

keep 15 servants working. It should be mentioned that the utilization percentages given refer to the actual ray tracing phase of the program only, i.e. time for initializing the master process, creating the servant processes, and reading the scene description file is not taken into account. However, for ray tracing complex scenes, initialization is of orders of magnitude smaller than time needed for computation.

5 Conclusion

In this paper, we have described how hybrid monitoring techniques supported the development of a parallel program for the SUPRENUM multiprocessor. Hybrid monitoring proved to be an excellent method for debugging and tuning of parallel programs on a novel machine. The example application is a ray tracer parallelized according to a dynamic ray partitioning scheme. Measurements of the instrumented program were carried out using the hardware monitoring system ZM4.

Event-driven hybrid monitoring with global timing information provided insight into the runtime behaviour of the parallel program which could not have been gained using other methods. We have shown how a basic implementation of the ray tracer could be improved significantly with the help of measurements. Efficiency of the program could be increased dramatically.

It would certainly be very interesting to measure the operating system and not only the application program. Instrumenting SUPRENUM's operating system to find more detailed information about the behaviour of the node scheduling algorithm and internode communication is one of our goals.

In our future work we intend to make use of SUPRENUM's vector processing capabilities. More precisely, we plan to implement a hierarchical bounding volume scheme based on parallelopipeds. Plane intersection operations will be vectorized to further increase the performance of the servant processes.

References

[1] T. Bemmerl, R. Lindhof, and T. Treml. The Distributed Monitor System of TOPSYS. In H. Burkhart, editor, *CONPAR 90–VAPP IV, Joint International Conference on Vector and Parallel Processing. Proceedings*, pages 756–764, Zurich, Switzerland, September 1990. Springer, Berlin, LNCS 457.

[2] A. Böhm, J. Brehm, and H. Finnemann. Parallel Conjugate Gradient Algorithms for Solving the Neutron Diffusion Equation. In *International Conference on Supercomputing*, pages 163–172, Cologne, June 1991 1991. ACM Press.

[3] H. Burkhart and R. Millen. Performance Measurement Tools in a Multiprocessor Environment. *IEEE Transactions on Computers*, 38(5):725–737, May 1989.

[4] P. Dauphin, R. Hofmann, R. Klar, B. Mohr, A. Quick, M. Siegle, and F. Sötz. ZM4/SIMPLE: a General Approach to Performance–Measurement and –Evaluation of Distributed Systems. In T.L. Casavant and M. Singhal, editors, *Advances in Distributed Computing: Concepts and Design.* IEEE Computer Society Press, 1992.

[5] W.K. Giloi. SUPRENUM: A trendsetter in modern supercomputer development. *Parallel Computing, North–Holland*, 1988(7):283–296, 1988.

[6] A.S. Glassner. *An Introduction to Ray Tracing.* Academic Press Limited, London, San Diego, New York, Boston, Sydney, Tokyo, Toronto, 1989.

[7] A.D. Malony, D.A. Reed, and D.C. Rudolph. Integrating Performance Data Collection, Analysis, and Visualization. In M. Simmons and R. Koskela, editors, *Performance Instrumentation and Visualization*, chapter 6, pages 73–98. ACM Press, Frontier Series, Addison–Wesley Publishing Company, New York, 1990.

[8] B.P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski. IPS–2: The Second Generation of a Parallel Program Measurement System. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):206–217, April 1990.

[9] A. Mink, R. Carpenter, G. Nacht, and J. Roberts. Multiprocessor Performance–Measurement Instrumentation. *Computer*, pages 63–75, September 1990.

[10] B. Mohr. SIMPLE: a Performance Evaluation Tool Environment for Parallel and Distributed Systems. In A. Bode, editor, *Distributed Memory Computing, 2nd European Conference, EDMCC2*, pages 80–89, Munich, Germany, April 1991. Springer, Berlin, LNCS 487.

[11] K. Peinze. The SUPRENUM preprototype: Status and experiences. *Parallel Computing, North–Holland*, 1988(7):297–313, 1988.

[12] J.-P. Richter. Parallelisierung des Ray Tracing Verfahrens. Internal study, Universität Erlangen–Nürnberg, IMMD VII, Januar 1992.

[13] K. Solchenbach and U. Trottenberg. SUPRENUM: System essentials and grid applications. *Parallel Computing, North–Holland*, 1988(7):265–281, 1988.

[14] J.J.P. Tsai, K. Fang, and H. Chen. A Noninvasive Architecture to Monitor Real–Time Distributed Systems. *Computer*, pages 11–23, March 1990.

[15] T. Whitted. An Improved Illumination Model for Shaded Display. *Communications of the ACM*, pages 343–349, June 1980.