MISS-PVM Extension for Simulating Dynamic Load Balancing

Helmut Hlavacs[1]
Dieter F. Kvasnicka[2]
Christoph W. Ueberhuber[3]

[1]Institute for Applied Computer Science and Information Systems,
University of Vienna
hlavacs@aurora.tuwien.ac.at

[2]Institute for Technical Electrochemistry,
Technical University of Vienna
kvasnicka@tuwien.ac.at

[3]Institute for Applied and Numerical Mathematics,
Technical University of Vienna
christof@uranus.tuwien.ac.at

January 1999

AURORA TR1999-02

Abstract

The *Machine Independent Simulation System for PVM 3* (MISS-PVM) makes it possible to develop software for parallel computers which are not available in reality. MISS-PVM can also be used as a tool to produce timing measurements of existing programs which are independent of actual load characteristics. MISS-PVM also makes the debugging of parallel programs easier. To exploit these features, it is not necessary to rewrite existing code or create additional code (for either C or Fortran). Programs utilizing PVM can be used without modification.

In this report, several new features added to MISS-PVM are described. The use of virtual messages simulates the effect of long messages while sending only small ones, thus increasing the simulation speed. Also, network contention is now included, simulating the concurrent access to shared networks. The new conservative parallel discrete event simulation protocol will also ensure the correct ordering of messages. Finally, the add-on package *Workstation User Simulator* (WUS) will now accept also real applications in addition to stochastic application models.

Contents

Chapter 1

Introduction

The increasing number of available parallel computers or clusters of workstations, interconnected with high speed networks, has created a need for efficient parallel software. Developing such is difficult due to the additional communication overhead often necessary. Factors influencing the efficiency are, for instance, the problem size, the percentage of sequential code, the speed of the communication system, the communication/computation ratio or the number and type of processors used. Parallel programs running efficiently on one parallel computer might be very inefficient on others.

There are several ways of comparing algorithms for parallel computers, all having drawbacks. Analytical models are very difficult to create, might be based on simplifying assumptions and often cannot catch the possibly complicated structure of the simulated environment or the parallel programs.

Comparison by executing the programs is restricted to available parallel computers only, the program behavior on different platforms, interconnected with different networks, cannot be obtained. Also, the execution of parallel programs might use up large amounts of CPU time, thus consuming computing time possibly needed otherwise.

Simulation tries to bridge the gap between those two approaches. Parallel programs are still mathematically modeled, yet the simulation environment allows to include features untreatable by mathematical analysis only. Also, the program behavior can be observed on platforms behaving differently, and choosing any number of processors.

MISS-PVM has been developed to simulate the execution of real code on different platforms, using different networks. The *Workstation User Simulator* WUS has been created to additionally simulate the effect of workstation users starting competing processes, making it also possible to run statistical models instead of real code, thus speeding up simulation runs by orders of magnitude.

This report consists of two parts. In Chapters 2 and 3, MISS-PVM will be described. Chapter 2 describes basic features, such as the relation of MISS-PVM to PVM3 and the concept of virtual time and virtual machines. Chapter 3 states some restrictions of MISS-PVM that must be kept in mind when using it.

Chapters 4 and 5 then describe the new features of MISS-PVM and WUS. In Chapter 4, basic enhancements of MISS-PVM such as virtual messages and the new synchronization protocol are denoted. Finally, in Chapter 5, the interaction between MISS-PVM and WUS is described, together with new features of WUS.

Chapter 2

Features of MISS-PVM

The *Machine Independent Simulation System for PVM 3* (MISS-PVM) makes the development of software for parallel computers which are not available in reality possible. And it can also be used as a tool to produce timing measurements of existing programs which are independent of actual load characteristics. MISS-PVM also makes the debugging of parallel programs easier. To exploit these features, it is not necessary to rewrite existing codes or create additional ones (for either C or Fortran). Code using PVM can be used without modification.

2.1 Overview

The above mentioned features are obtained by using a virtual layer which, once established, does not have to be tampered with when developing software. The *Virtual Layer for PVM 3* is situated between the user program and PVM 3. It redirects all PVM 3 subroutine calls to itself, performs certain virtual timing and virtual machine adaptations and sends the calls (in a modified form) to PVM 3.

In order to use the virtual layer it is not necessary to modify the user programs. It is only necessary to use different include files and to link the program to additional libraries. The virtual layer generates output files which trace calls to communication subroutines. This tracefile is intended to serve as input for post-mortem visualization.

This method has two major advantages over normal trace file writing: the *Virtual Layer for PVM 3* (i) uses its own simulated *system time* and (ii) makes a *virtual machine* available to the user.

This virtual machine can simulate a wide variety of machines, which can be either non-existent or not available at the moment. Machine parameters are read from a file when the program starts. These parameters may also be changed dynamically during the program execution.

The virtual layer makes it possible to compare program runs on computers with different communication latency and computation speed (independent of actual load characteristics). Time-measurement of different load balancing strategies can be made quickly and enables the determination of the optimum strategy for certain architectures[1].

2.2 Using PVM 3 with MISS-PVM

PVM 3 is a software system linking a network of Unix computers in such a way as to create a single large (parallel) computer. It provides message passing and process control routines for tasks which run on different computers. PVM 3 uses daemon processes on every host[2] to establish communication from one user process to another one. The user processes can send their messages only to the PVM daemon on their actual machine, the PVM daemon communicates with the PVM daemon of the target machine, and this PVM daemon delivers the message to the receiver. User programs are linked to the PVM 3 library, which contains routines serving as interfaces to the pvmd.

Figure 2.1 shows the relationship between the parallel user program, PVM 3, and the native communication primitives available on various computer systems. The user program calls PVM 3 subroutines in order to pass messages between different processes and in order to create and to terminate processes on various network nodes. The PVM 3 subroutines in turn perform their respective tasks by calling the native communication primitives of the underlying computer system. In this way a user program can be run without modification on a variety of different computer systems, as the use of PVM 3 subroutines makes the native communication primitives transparent.

In Figure 2.2 a new level between the user program and PVM 3 is added: the *Virtual Layer for PVM 3*. It uses only PVM 3 calls and contains no machine dependent routines. So it is obvious that the *Virtual Layer for PVM 3* can run on a wide variety of machines (like PVM). This library also provides other advantages by making virtual time, virtual machines, and output generation for fully graphical post mortem visualization available. The user program as well as the PVM 3 level remain unchanged. The only difference is that an include file redirects PVM 3 routine calls. The virtual layer creates output which is post processed by simple programs and is then used as input for post mortem visualization with ParaGraph.

So there are three steps in every program run:

Execution of the parallel program. Two modifications have to be made within programs: An include statement has to be changed and an additional library has to be linked to the program. Calls to a PVM 3 routine should

[1]For short execution times stochastic effects may possibly overlap measurement data due to computer timing routines that are too coarse grained.

[2]The PVM daemon is often shortened to `pvmd` or `pvmd3`.

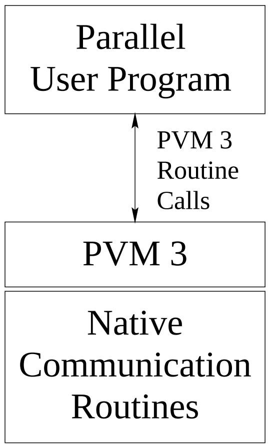

Figure 2.1: Position of PVM.

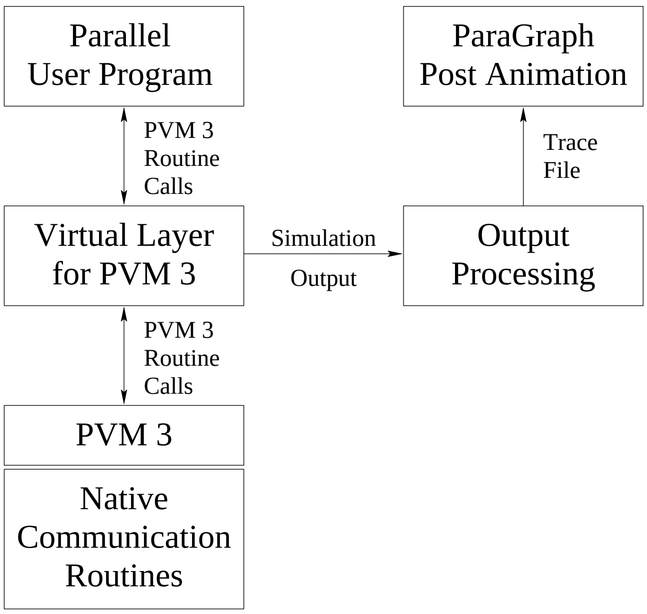

Figure 2.2: Virtual Layer for PVM.

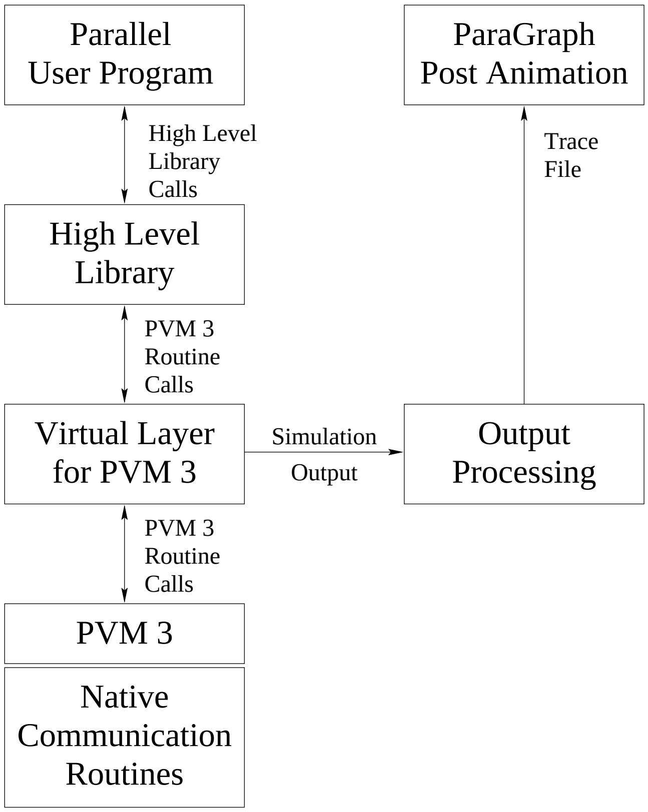

Figure 2.3: High level communication library, e. g., the basic linear algebra communication subroutines (BLACS), used with PVM 3.

take place at the beginning of every program being executed in parallel, so that correct timing is ensured. Normal timing routines should not be called, because normal system time routine calls do not return the simulated system time (with the exception of the `time()` call, which is also redirected to MISS-PVM. `pvmS`-calls (see Section 2.4) can be added in order to use special properties of the MISS-PVM.

Calling the output processing program. This is done by calling a script file after the program execution in order to collect the output files of all tasks running in parallel and to prepare these output files to be read by ParaGraph. In order to write the output to the output file the script file has to have the name of its output file as a parameter. The script file then adds the extension `.trf` to this name.

Visualizing the output using ParaGraph. ParaGraph takes the full name of the output file (with the extension `.trf`) as parameter. Visualization takes place in ParaGraph via various animation windows, which are, to a great extent, self-explanatory (see Tomas, Ueberhuber [15], Heath [8]).

2.3 Communication Libraries and MISS-PVM

Once the high level library routines (based on PVM) have been recompiled, no additional modifications are necessary. The user program is linked with the new library. The result is a program that performs the same as before the library has been recompiled, except that it writes an output file and can be simulated on virtual machines.

In Figure 2.3 a new level is added to the diagram depicted in Figure 2.2. A high level library is situated between the user program and the *Virtual Layer for PVM 3.* The user program remains unchanged when the high level library is recompiled, and the virtual layer works with the routines in the same way it works with normal user programs.

When working at a higher level of abstraction than PVM 3, not everything that can be viewed with ParaGraph is easy to understand. Post mortem visualization should be limited to general information windows like those that show the communication/computation ratio.

2.4 Virtual Time

The *Virtual Layer for PVM 3* uses an internal time that is based on three components.

Computation Time which is actually used by executing user programs.

Communication Time which is calculated using the parameters of the virtual machine.

Waiting Time according to parallel execution.

These three components are added to give the "virtual time" of each user process.

The internal timing granularity of the virtual time is 100 microseconds (or sometimes 1 microsecond). Time output is always measured in microseconds. The additional two digits have to be inserted, because in ParaGraph the input has to be sorted very exactly according to time. No two time stamps can be the same for each task.

2.5 Virtual Machines

Virtual Machines are defined in the file `VMConfig`. The format of this file is as follows: In the first line are the parameters of a machine which is used for the master program and as default for all programs started without an explicit name for the machine or the host type given. In the other lines comments (beginning with the symbol `#`), or machine or host type specifications can be found.

Parameters found on specification lines are:

Name of the Machine or Host Type. This name is used in `pvm_spawn`. If the following parameter is 0, the machine is assumed to be "real", and the program is started on this machine. Otherwise the machine has a "virtual" name, and PVM 3 is asked to look for a suitable machine.

Performance Factor. This is a floating point multiplier for computation time. If this parameter is 0, the machine name is sent to PVM 3 and the computation timing results are not changed.

Initialization Delay. This is the time[3] needed for `pvm_spawn` to be called.

Send Delay. This is the time[3] used for sending a message using `pvm_send` or `pvm_mcast`. This time contains packing the message, resolving the address of the host and starting the transmission (as far as the sending process is involved). This time is independent of the message length.

Receive Delay. This is the time[3] used in calling the receive routines `pvm_rcv`, `pvm_nrecv` and `pvm_probe`. This time is always the same whether these routines succeed or fail.

Transmission Delay. This is the time[3] used to transfer a message which is independent of the message length.

[3]Times units are measured in increments of 100 microseconds.

Transmission Proportional Delay. This is the time[3] used for the transfer of a message which depends on the message length. This time is measured in 100 microseconds per kilobyte.

These parameters are used in the performance modeling file `delay.c`. If the actual performance model turns out not to be exact enough, it can be easily be changed by modifying this file.

2.6 Transparency

During the development of the *Machine Independent Simulation System for PVM 3* it was crucial to maintain all PVM 3 functions. When PVM 3 is used with the virtual layer, virtual time routines and virtual machine properties are added; otherwise there are no other modifications (cf. Sections 2.4, 2.5, 3).

Virtual time routines should replace system time routines. This is because system time calls can only return real time or process time, which is inappropriate for the user program's time. Virtual time is calculated from process time, but it is modified by adding certain delays from inter process communications and by subtracting overhead time that results from simulation using the virtual layer.

Using the virtual layer entails the simulation of virtual machines, which leads to a situation in which it is not clear which computer actually executes user processes. So user programs should not try to establish perfect communication patterns by determining their actual host.

This unpredictability in determining the actual host results in the following situation: If programs are started on a cluster of inhomogeneous computers, time-measurements will not be reliable, since one machine will execute a certain code faster than another machine. So there are two solutions: The first one is to run all program instances on the same type of machines (a homogeneous cluster) or on a single machine. MISS-PVM spawns a new task on a machine that was selected by PVM 3 if either no machine name is given in the call to `pvm_spawn` or if the machine performance factor in the file VMConfig has a value different from 0. The second (and much more difficult) solution is to spawn processes on different machines which are described in the virtual machine configuration file `VMConfig` (see Section 2.5). Spawning a new task onto a virtual host is done by using the virtual hostname (the first parameter of a line in the file `VMConfig`) as hostname and by setting the flag parameter to `PvmTaskHost` (1) or `PvmTaskArch` (2).

Chapter 3

Restrictions of MISS-PVM

3.1 Restrictions due to Execution Time

For large programs (especially those which perform compute intensive tasks computing) execution time can become a prohibitive factor. It is a good idea to begin with small parameters for problem dimensions (like the dimension of a matrix or the number of iterations). Not only the execution time for a program slows down computation but also the amount of memory that is used. If there are more parallel running processes than computing nodes, each process running in parallel on a single node will use much memory. This can lead to performance degradation if the machine starts to swap memory to the hard disk.

Many difficulties that arise during the construction of parallel programs do not need large scale problems to be eliminated, they can also be solved with small scale problems. ParaGraph is not easy to use with input files that are much bigger than a megabyte. This is due to one of two reasons. The first is that too much information is given to the user in too short a time, which leads to an animation that does not give enough detailed information. The second is that it can take too long to get to that part of the simulation which is of interest.

3.2 Internal Restrictions

When using the *Machine Independent Simulation System for PVM 3* the user has to deal with certain internal restrictions. The use of MISS-PVM will not be prevented by any of these restrictions, but some differences to the use of PVM 3 without MISS-PVM can result. Normally the user will not notice these differences. However, if unexpected results occur without explicit error messages they can come due to one of the following restrictions. Error messages may be hard to find, because the output of child tasks does not always go to the user terminal. In these cases the user has to look for messages in the file `/tmp/pvml`.uid (uid is the user ID).

- The master task must not be called from the PVM 3 command line. It would wait forever for initialization data, because the `pvm_parent()` routine behaves different in this case.
- `pvm_mytid` (or any other PVM 3 routine) has to be called as soon as possible after the task starts (in order to generate correct timing measurements). After the master task has sent a message to its children the childrens' clocks are automatically synchronized with the master clock.
- `int` must be 32 bit wide in order to ensure correct timing. Program runs longer than 200000 seconds produce a timer overflow[1].
- Group routines are not supported by the virtual layer.
- All hosts in the cluster must have the same performance characteristics in order to produce reliable computation timing output.
- Programs which use the virtual layer are only allowed to run in parallel once for each user. This is because during synchronization they communicate with every task spawned by the same PVM 3 daemon. This sort of synchronization is used in `pvm_nrecv` and `pvm_probe` routine calls.
- The length of messages sent is not reported (because this is not supported by PVM 3). It is reported, however, after a receive statement.
- Empty `pvm_nrecv`-loops may run a very long time, because every call of this routine produces a line (or two) in the output file. While these lines are being written, the simulation time is advanced very slowly.

3.3 Restrictions in the Use of Specific Routines

There are complications involved with using certain PVM 3 routines adapted to the virtual layer. These routines are modified, but do not return error conditions if there is an error. The complete list of these routines is as follows:

- `pvm_kill` — in situations where error conditions are returned, it is possible that return values are incorrect. The reason for this is that it is not possible for one task to kill another in the virtual layer. The task to be killed might not have an advanced enough virtual time to be killed. What happens instead is that a message is sent to the task to be killed to "commit suicide" at a given time. The "killer task" does not wait for the other to "commit suicide". This is why the "killer task" does not return the correct value.
- `pvm_tasks` — the virtual host computer is not reported correctly.

[1] Internal time granularity is $10^{-4}s$.

- `pvm_mstat` — the status of virtual hosts is not reported.
- `pvm_config` — could give more information about all virtual hosts, but it is not supported yet.
- `pvm_recvf` — due to the internal restructuring of receiving statements and the hope that nobody will ever need this routine when using MISS-PVM, the virtual layer does not support it correctly.

Chapter 4

Load Balancing Simulation using MISS-PVM

Dynamic load balancing strategies (Vaughan, Donovan [16], Heirich, Arvo [9]) redistribute the computational load at runtime, thus sending messages of various lengths across the network. The time to send and receive such messages will have a considerable effect on the efficiency of the strategy of choice. When simulating load balancing strategies, it is therefore important to include the communication time, depending on the speed of the communication network and the length of the messages sent. For a given communication network, simulation runs then can be done for a large number of different message lengths, yielding quantitative and qualitative information about the redistribution process and about the dependence between message lengths and the effectiveness of the strategy under consideration.

Consider for example a master-slave scheme, where a master sends work to waiting slaves. The master sends work requests to the previously spawned slaves. On receiving these requests, the slaves start their work and may send intermediate results to neighbors. If a result is finished, it is sent back to the master. After all results have arrived, the master sends further work requests to its slaves, until all the work is done. Such a work scheme is used, for example, in the parallelization of the VISTA ion implantation module. VISTA (Strasser et al. [13], Grasser et al. [6]), developed at the Institute for Microelectronics at the Technical University of Vienna, is a framework for the design and simulation of process steps involved in semiconductor production. VISTA includes an ion implantation program (Bohmayr et al. [2]), which is based on a Monte Carlo simulator computing the endpoints of ions shot into a substrate. The resulting ion density is needed later on to predict the electrical behavior of the investigated semiconductor.

Figure 4.1 shows the space-time diagram of the simulated parallel version (Hlavacs, Ueberhuber [10]). The parallel version of the ion implantation will be run on an interactively used heterogeneous workstation cluster. On such a cluster,

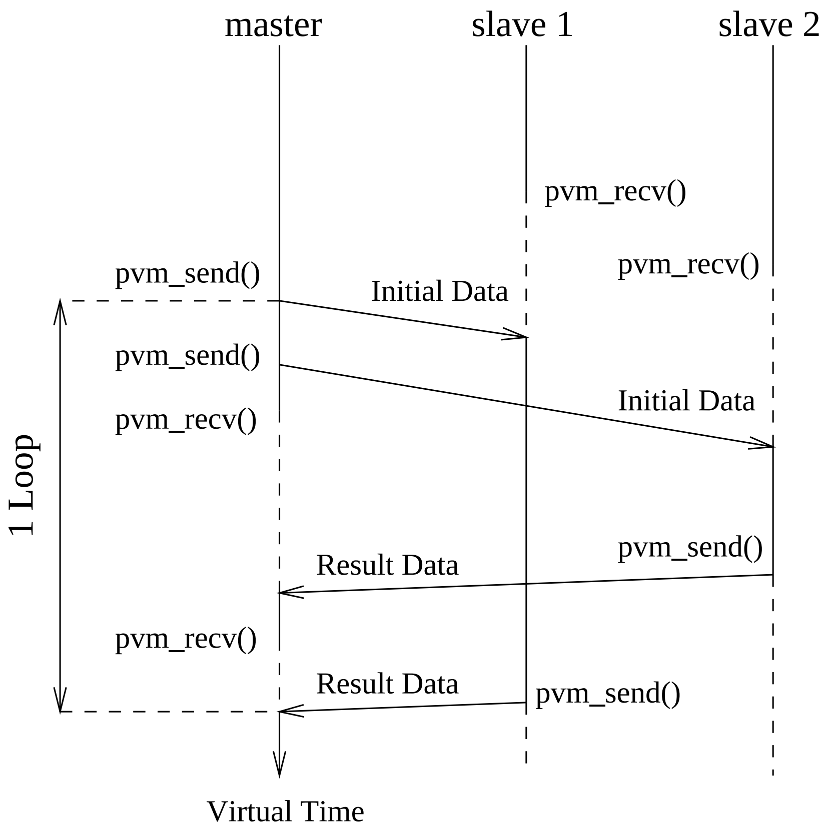

Figure 4.1: Master-Slave Behavior.

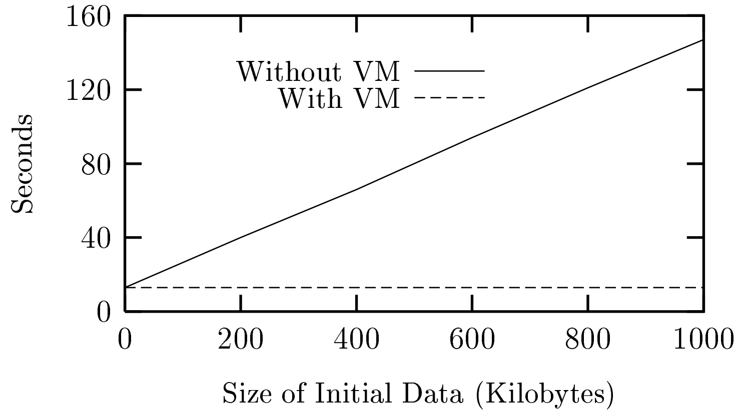

Figure 4.2: Effect of virtual message length (VML).

other users might start processes of their own, thus taking away processor time and getting the ion implantation program out of balance. The goal of simulation is thus finding efficient ways of reacting to workload changes and rebalancing the implantation program at runtime.

4.1 Virtual Message Lengths

If we want to examine the effect of different message lengths for the *work request messages* sent by the master, the original version of MISS-PVM would force the master to generate PVM 3 buffers of the examined message length and send it to each child. The children would receive the buffers and decode them. If the simulation is carried out on one single computer and the message length is quite large (many MBs), the main computer storage is likely to run out and the computer will be forced to start swapping, thus increasing the time necessary for simulation accordingly.

When simulating load balancing algorithms by using application program models instead of the real applications, it is not necessary to actually create the PVM 3 buffers, it is sufficient to know the message length and tell it to the simulation environment. MISS-PVM thus has been enhanced to accept such *virtual message lengths*. Passing the virtual message length to MISS-PVM now can be done by using the external variable `pvmS_OverrideMessageLength`. If this variable is set to a value less than zero (which is initially the case), MISS-PVM uses the length of the PVM 3 message buffer used. If `pvmS_OverrideMessageLength` is set to $n > 0$, then the communication time for this message is calculated on the basis of n bytes. Fig. 4.2 shows the effect of using the virtual message length instead of real PVM 3 buffers. In this example, the parallel version of VISTA is simulated on a 166 MHz Pentium with 64 MB main memory on a Linux operating system. It is assumed that 20000 ions are implanted, using 10 slaves. The internal effort for memory management increases linearly with the size of the used PVM 3 buffers. By using virtual message lengths, the effort is kept constant.

Additionally, larger message buffers could not be used, as PVM 3 would not be able to allocate memory for its buffers and would exit the simulation.

4.2 Including Network Contention

MISS-PVM considers the network to yield the same capacity to each message sent, not including the effect of overlapping messages impeding each other. If two processes, for instance, send messages at the same time on an ethernet network, each message needing n seconds for transfer, then the first message would block the second one for additional n seconds. This has not been considered in the original version of MISS-PVM.

Such a behavior now has been included into MISS-PVM by implementing a conservative parallel discrete event simulation protocol based on an extra process called daemon. Each PVM 3 process having no parent (usually the master process) starts this daemon and later on passes the daemon pid to its children. This way, a master shares a physical communication line with its children.

The daemon keeps an internal task list of all running PVM 3 processes. Upon receiving messages, the daemon updates its task list by calling a modified version of `pvm_tasks()`, which will not return the daemon pid. Each entry in this list can have one of the following states:

Unknown: The PVM 3 process is believed to do work.

Waiting for line: The process has called `pvm_send()` (the MISS-PVM version). By calling this function, the process sends a MISS-PVM virtual layer message to the daemon, who in turn updates the state of the process.

Waiting for probe: The MISS-PVM versions of `pvm_probe()` or `pvm_nrecv()` have been called. This way, a virtual layer message is sent to the daemon.

Blocked receive: The process has called the MISS-PVM version of `pvm_recv()` and is waiting for messages.

Deleted: In this case, the process is removed from the task list and is added to a deletion list.

Each new task list entry is initially marked *unknown*. If a process wants to send a message to another process, it first sends a virtual layer message to the daemon and waits for a reply from the virtual layer. The daemon, upon receiving the first message, sets the sending process to the state *waiting for line*. Then, the daemon checks, whether the state of all other PVM 3 processes in its list are known (either *waiting for line*, *waiting for probe* or *blocked receive*). If this is not the case, the daemon waits for further messages from the missing processes, until finally all have sent their states. If a process exits, it sends an appropriate message to the daemon, who removes the process from its task list and adds it to the deletion list.

Once, the states of all other processes are known, the earliest sender (according to the virtual time) is granted the virtual physical communication subsystem and may proceed. This is done by sending a virtual layer message to the *receiver*, containing information about the message size and the sender pid. The receiver finally sends a wake up message to the sender, who then sends its message data to the receiver. The protocol thus needs a total of three virtual layer messages with fixed size and one user data message arbitrary size.

After sending the virtual layer message to the receiver, the sender's and receiver's states are once again set to *unknown* by the daemon.

The new protocol for one master and two slaves can be seen in Fig. 4.3. Fig. 4.4 shows the effect of neglecting and considering network blocking on the simulation result. In this example, the effect of increasing the length of sending the initial data (see Fig. 4.1) on the simulation result is demonstrated.

4.3 Removing Synchronization Errors

The new protocol has also the advantage of removing synchronization errors observed in the previous MISS-PVM version when executing non-deterministic application programs. Such errors occurred, in case some processes would be slowed down by other processes running on the same physical machine, started by other users. Such processes would receive smaller amounts of CPU time on their machines, and as a result, their virtual time would advance more slowly, as in MISS-PVM, the amount of CPU time received controls the process's virtual time. Fig. 4.5 shows such a synchronization error. Both P_1 and P_2 want to send to P_3. When sending the message to P_3, let the virtual time of the processes be $v_1 = 300$ and $v_2 = 200$, respectively. As sending the messages needs 5 virtual time units, the message of P_2 should be received first at virtual time point 205. Also, let there be competing processes on P_2's machine. In real time, P_1 will be granted 300 CPU seconds first, and P_2 its 200 CPU seconds last (due to the competing process). P_1 then will send its message first to P_3, who is in blocking receive. As P_3 will get the first message, it will set its virtual time to 505, not knowing that it should have received the message of P_2 at virtual time 205! After reacting to this message, P_3 will again wait for messages in a blocking receive, by calling `pvm_recv()` at some virtual time $v_3 \geq 305$. Finally, P_2 will send its message at its virtual time 200. P_3 will then receive a message that it should have received at virtual time $205 < 305$, before the first one. As P_3 cannot unroll its most recent activities, the message from P_2 is then regarded as received at time 305 instead of 205! The observed errors can have an influence on the critical

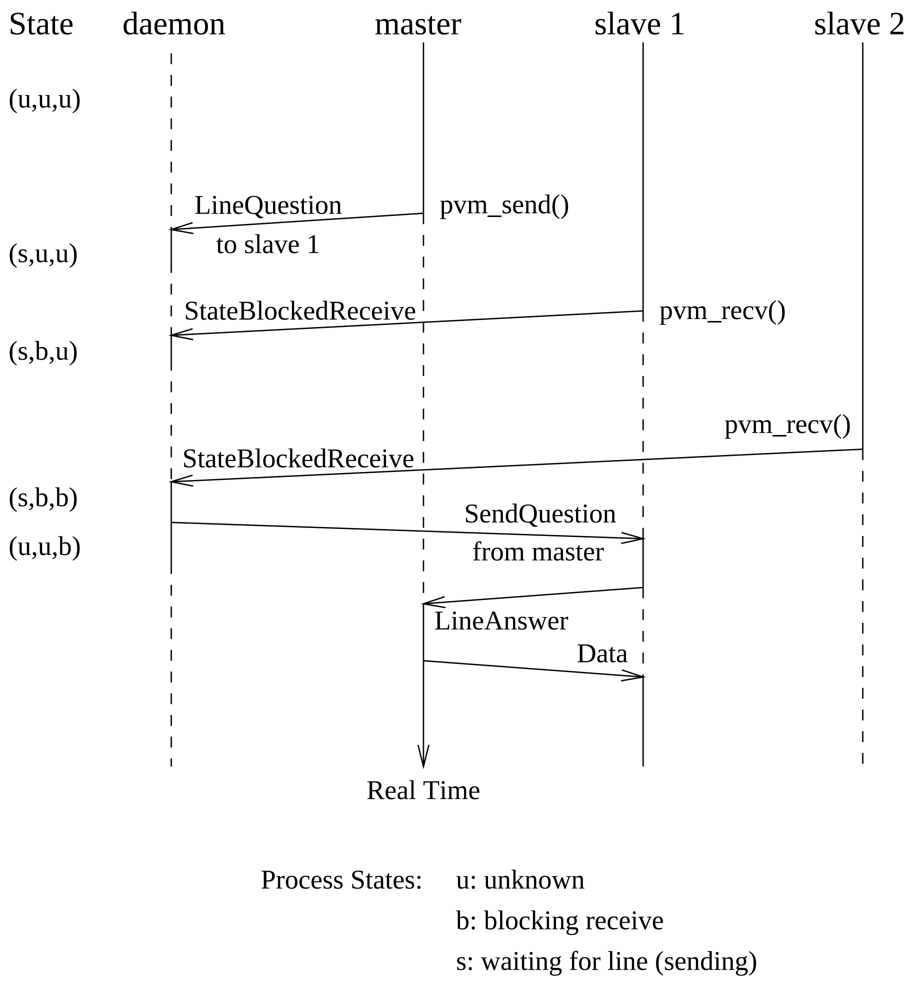

Figure 4.3: New MISS-PVM send protocol.

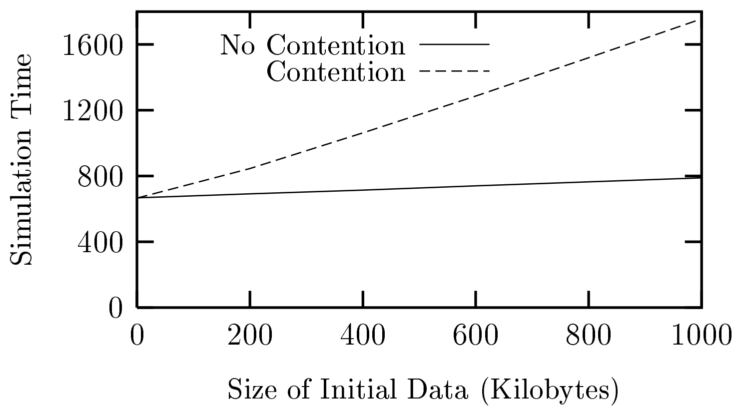

Figure 4.4: Effect of considering network contention on simulation result.

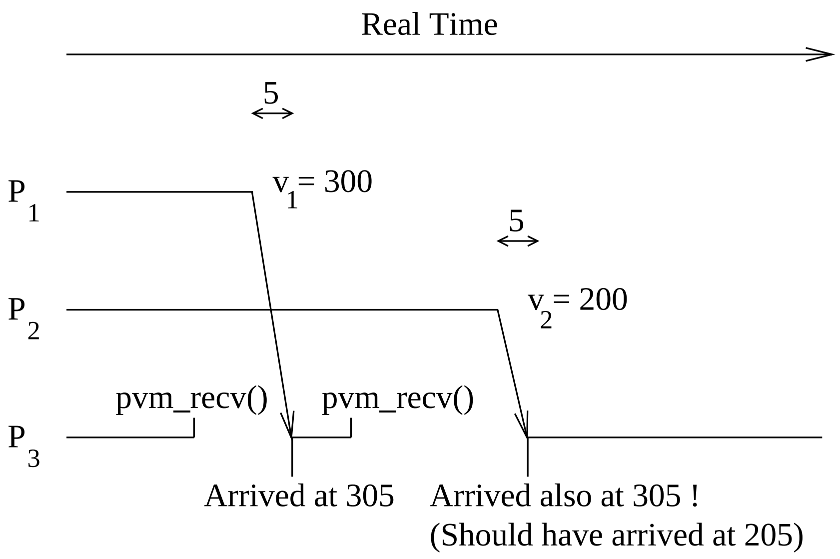

Figure 4.5: Synchronization error in previous MISS-PVM version.

path, thus yielding incorrect results.

In the new protocol, all sends are serialized by the daemon. It is assumed that only one process can have access to the communication subsystem. Thus, the order at the receiver's end is preserved and the above described synchronization errors cannot happen.

Chapter 5

Competing Processes

The *Workstation User Simulator* WUS (Hlavacs, Ueberhuber [10]) is an add-on to MISS-PVM. WUS simulates the generation of competing processes, running in parallel on interactively used workstation clusters, and taking away CPU cycles there. Processes can be generated by using fixed arrival and departure rates, variable arrival and departure rates provided by tracefiles (Calzarossa, Serazzi [3]), tracefiles of real processes (Zhou [17]) and user behavior graphs (Calzarossa, Serazzi [4]).

By constructing stochastic models of real parallel applications or running real applications, different load balancing schemes can be simulated and compared with each other. The structure of the whole simulation system can be seen in Fig. 5.1. It is important to note that the competing processes are not started in reality, but are only represented by list entries in the virtual CPU (VCPU) queue of WUS. The WUS VCPU is, however, tightly linked to the MISS-PVM virtual time. Whenever a WUS process consumes VCPU time, WUS increases the MISS-PVM virtual time accordingly.

5.1 WUS Scheduling

The application model calls WUS functions to state that it wishes to be granted n VCPU seconds. The new WUS version now schedules its virtual CPU to all running process by using *priority scheduling* as implemented in the Linux kernel, driven by the standard UNIX nice levels. The Linux process scheduler is implemented in the Linux kernel source file `sched.c`. Mapping the nice level $-20 \leq n \leq 19$ to the used priority $1 \leq p \leq 40$ is implemented as in `sys.c`.

Like in the *processor sharing* paradigm (Allen [1]) in queuing systems, it is assumed that the time-slices scheduled to each process are infinitely small (in contrast, the duration of each time-slice on an i386 Linux system is 10 ms). If there are, for example, two processes P_1 and P_2 competing for the VCPU, each being assigned the priorities p_1 and p_2 respectively, then P_1 will consume

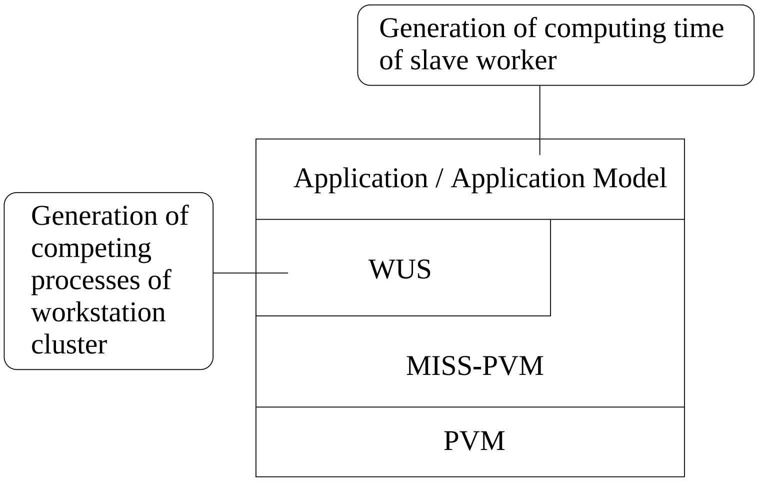

Figure 5.1: Structure of the simulation system.

nice level	Lin (1)	WUS (1)	Lin (2)	WUS (2)	Lin (3)	WUS (3)
0	50.0	50.0	33.2	33.3	24.9	25.0
5	42.6	42.9	28.0	27.3	20.4	20.0
10	33.4	33.3	19.9	20.0	14.5	14.0
15	20.0	20.0	12.9	11.1	8.5	7.7
19	7.5	4.8	3.3	2.4	2.4	1.6

Table 5.1: Percentages of granted CPU time, depending on the used nice level and the number of competing processes (inside the brackets). *Lin* denotes the measured values on the Linux computer, *WUS* denotes the percentages as granted by WUS.

$p_1/(p_1+p_2)$ VCPU seconds per second, and P_2 the remaining time.

Table 5.1 shows measurement results from a real computer (Linux operating system with kernel 2.0.36, on a Pentium II 400 MHz) and the corresponding WUS scheduling. The results were obtained by starting one, two and three competing processes additionally to the observed process. The percentage of granted CPU time was then collected by using the standard UNIX command `top`, showing the CPU percentages of all running processes. All percentages were normalized with the sum of the observed processes only. As can be seen, there is a good match for commonly used nice levels[1]. The largest error is observed for the maximum nice level 19, possibly because of the Linux scheduler giving the currently selected runnable process a little advantage.

[1]Calling `nice` *`cmd`* without parameter `-n` starts the program *`cmd`* with nice level 10.

5.2 Load Balancing Simulation using WUS

The main parts of the simulation system consist of WUS and MISS-PVM. MISS-PVM provides a virtual layer over PVM 3 together with a virtual time depending on the process CPU consumption. WUS will generate competing processes, driving the simulated parallel program out of balance.

In order to simulate load balancing algorithms under different load situations, the first version of WUS required a stochastic model of the real application.

Usually, a parallel program consists of two or more processes communicating with each other. This simulation system assumes communication with PVM 3 calls, each call being replaced by its MISS-PVM call. Each process will then either do some CPU intensive computation[2] or communicate with another process by sending and receiving messages.

The first version of WUS was only able to use stochastic application models to drive the simulation. The advantage of stochastic application models lies in their lack of consuming real CPU time. The simulation speed is thus increased drastically, making it possible to simulate days of parallel computation within seconds on one workstation only.

Simulating VCPU consumption was done by letting the stochastic application model call the `RunProcess( cputime )` member function of an instance of the WUS class *Computer*, making WUS to generate another process in its VCPU queue and to continue simulation of VCPU consumption, until the new process has consumed `cputime` VCPU seconds.

After this, WUS would increase the MISS-PVM virtual time and would return to the calling application model.

5.3 Connecting MISS-PVM with WUS

The new version of MISS-PVM calls WUS member functions over a C wrapper function, and allows to run *real applications* instead of stochastic application models only. The C wrapper is activated upon the construction of an instance of the WUS class *Computer*, and deactivated upon its destruction. Thus, as before, MISS-PVM can be used without WUS. Fig. 5.2 shows the sequence of calls.

First, the real application calls a PVM 3 function, which is replaced by the according MISS-PVM call. MISS-PVM manages the virtual time and might send virtual layer messages to other processes. This is then done by using PVM 3. Also, MISS-PVM saves the amount of CPU time, this UNIX process has consumed so far.

The real application will then consume n CPU seconds and, in order to do some communication, will finally make a PVM 3 call again, again being replaced by the according MISS-PVM call. MISS-PVM will then detect that there is an n

[2] Simulating I/O accesses is not supported in this simulation environment

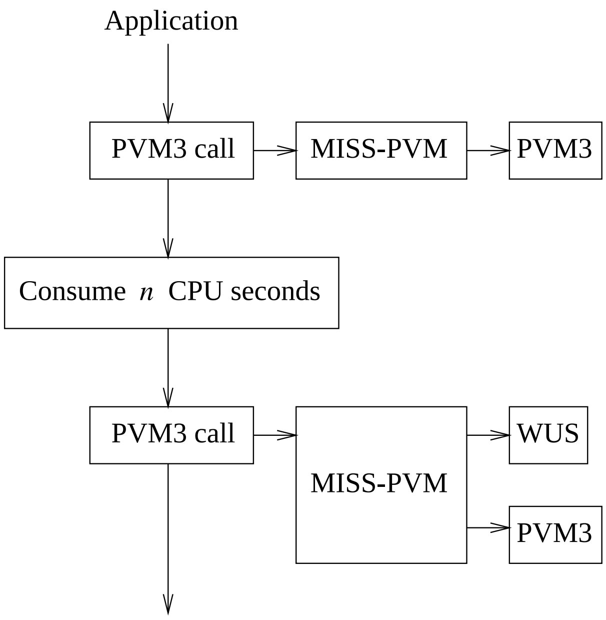

Figure 5.2: Real application consuming CPU time.

seconds difference in the CPU time to the last call. Instead of using n directly to increment its virtual time, the new version of MISS-PVM will call WUS instead and will use WUS to simulate, how many seconds more would have been needed, if there were competing processes fighting for the CPU. After this, WUS will then increase the MISS-PVM virtual time according to this new result, and on return, MISS-PVM will perform the requested task by calling PVM 3 directly.

Chapter 6

Conclusion

In this report, several modifications and enhancements of MISS-PVM and WUS have been described. The creation of virtual messages allows to significantly speedup simulation runs. A new conservative parallel discrete event simulation protocol allows to include network contention into the simulation. As a byproduct, synchronization errors occuring when simulating non-deterministic application programs are removed.

By linking MISS-PVM with WUS, no longer only statistical application models, but also the real applications themselves can be used for simulating load balancing on heterogeneous, interactively used workstation clusters. Also, WUS has been changed to use *priority scheduling* instead of *processor sharing*. Processes can be started by using the standard UNIX nice value.

Acknowledgments

We would like to thank Siegfried Selberherr, Erasmus Langer, Mustafa Radi and Andreas Hössinger (Institute for Microelectronics, Technical University of Vienna) for their cooperation.

In addition, we would like to acknowledge the financial support of the Austrian Science Fund FWF.

Bibliography

[1] Allen A.O., *Probability, Statistics and Queuing Theory*, Academic Press, Orlando, 1990.

[2] Bohmayr W., Burenkov A., Lorenz J., Ryssel H., Selberherr S., *Monte Carlo Simulation of Silicon Amorphization During Ion Implantation*, Proceedings SISPAD 96 Conf., (2. - 4. September 1996, Tokyo), pp. 17-18.

[3] Calzarossa M., Serazzi G., *A Characterization of the Variation in Time of Workload Arrival Patterns*, IEEE Transactions on Computers C-34-2 (1985), pp. 156-162.

[4] Calzarossa M., Serazzi G., *System Performance with User Behavior Graphs*, Performance Evaluation 11 (1990), pp. 155-164.

[5] Krommer A., Ueberhuber C., *Dynamic Load Balancing—An Overview*, Technical Report ACPC/TR 92-2, Austrian Center for Parallel Computation, Vienna, 1992.

[6] Grasser T. et al., *VISTA Status Report*, Technical Report AURORA TR1997-16, Technical University of Vienna (1997).

[7] Halama S. et al., *The Viennese Integrated System for Technology CAD Application*, in Technology CAD Systems (F. Fasching, S. Halama, S. Selberherr, eds.), Springer-Verlag, Vienna, 1993, pp. 197-236.

[8] Heath M.T., *Recent Developments and Case Studies in Performance Visualization using ParaGraph*, in Performance Measurement and Visualization of Parallel Systems (G. Haring and G. Kotsis, eds.), Elsevier Science Publishers, Amsterdam, The Netherlands, 1993, pp. 175-200.

[9] Heirich A., Arvo J., *A Competitive Analysis of Load Balancing Strategies for Parallel Ray Tracing*, The Journal of Supercomputing 12 (1998), pp. 57-68.

[10] Hlavacs H., Ueberhuber C.W., *Simulating Load Balancing on Workstations with Irregularly Fluctuating Capacity*, AURORA Technical Report TR1998-11, Technical University of Vienna, 1998.

[11] Kvasnicka D. F., Ueberhuber C.W., *Simulating Architecture Adaptive Algorithms with MISS-PVM*, AURORA Technical Report TR1997-16, Technical University of Vienna, 1997.

[12] Kvasnicka D. F., Ueberhuber C.W., *Developing Architecture Adaptive Algorithms using Simulation with MISS-PVM for Performance Prediction*, Proceedings of the 1997 ACM/SIGARCH International Conference on Supercomputing, Vienna, Austria, July 7-11, 1997, pp. 333-339.

[13] Strasser R., Pichler Ch., Selberherr S., *VISTA—A Framework for Technology CAD Purposes*, Proceedings European Simulation Symposium, (19.-22. October 1997, Passau), pp. 445-449.

[14] Sunderam V.S., Geist G.A., Dongarra J., Manchek R., *The PVM concurrent computing system: Evolution, Experiences and Trends*, Parallel Computing 20-4 (1994), pp. 531-545.

[15] Tomas G., Ueberhuber C.W., *Visualization of Scientific Parallel Programs*, Springer-Verlag, Heidelberg, 1994.

[16] Vaughan J. G., O'Donovan M., *Experimental Evaluation of Distributed Load Balancing Implementations*, Concurrency: Practice and Experience 10-10 (1998), pp. 763-782.

[17] Zhou S., *A Trace-Driven Simulation Study of Dynamic Load Balancing*, IEEE Transactions on Software Engineering 14-9 (1988), pp. 1327-1341.