PSPVM2 – PVM for ParaStation

Patrick Ohly, patrick.ohly@stud.uni-karlsruhe.de

October 21, 1997

Abstract

This document describes the concept, implementation and performance of PSPVM2, the second port of the Parallel Virtual Machine (PVM, [2]) to the ParaStation system[1]. It is derived from the original PVM source code to retain compatibility with existing PVM applications and implementations, but uses a low level interface to the ParaStation system and direct communication within the ParaStation cluster.

PSPVM is the fastest PVM implementation on a workstation cluster with a process-to-process latency as low as $11.5\mu s$[2] and a sustained transfer rate of up to 9.4MB/s. It can still be used as part of a larger virtual machine and all PVM programs can be compiled for PSPVM without any changes.

1 Introduction

ParaStation is a communications fabric for connecting off-the-shelf workstations into a supercomputer. The fabric employs technology used in massively parallel machines and scales up to 4096 nodes. It offers a latency of $2\mu s$ and a throughput of 15.5MB/s. ParaStation's user-level message passing software preserves the low latency of the fabric by removing the operating system from the communication path, while still providing full protection in a multiprogramming environment (for more information see [1]).

The ParaStation programming interfaces comprise a UNIX socket emulation and a custom message passing environment. The custom interface supports task management and connection-less communication. Latency at this level is as low as $9.3\mu s$ and bandwidth as high as 9.4MB/s.

The Parallel Virtual Machine (PVM) is a software package for developing parallel programs executable on networked Unix computers. It turns a heterogeneous collection of workstations and supercomputers into a single high-performance parallel machine. It was originally designed for high latency networks but became a standard for fast parallel computers, too.

The first port of PVM to the ParaStation system (from now on called PSPVM) [1] simply replaced the system sockets with the ParaStation socket emulation. Used with system sockets PVM accounts for $154.5\mu s$ of the total $266.7\mu s$ latency, i.e. adds 137% delay. For ParaStation sockets this overhead becomes really significant due to the low latency of the ParaStation sockets of only $9.2\mu s$: now PSPVM causes an overhead of $53.5\mu s$ which are *580%!*

[1] http://wwwipd.ira.uka.de/ParaStation

[2] All measurements where done on a Alpha 21164, 500MHz.

This shows that the bottleneck is no longer the communications layer. For high speed communication fabric PVM has to be modified to reduce the latency caused by the inefficient implementation of its communication path.

This was the motivation for a new port with a different approach. The new version should feature:

1. High performance within the ParaStation cluster, especially regarding latency, but also bandwith.

2. Full compatbility with the original PVM application interface.

3. Using a ParaStation cluster together with other machines in one virtual machine should still work as usual.

In case of a conflict between these goals, performance should have priority, but compatibility to applications and other systems was still regarded as very important. As the result shows all of the goals could be reached with very little tradeoff.

2 Concept

2.1 ParaStation Cluster

The ParaStation offers several functions to treat a cluster of workstations connected with the ParaStation fabric as one computer. It is possible to spawn tasks, send signals to them and keep track of their status. Arbitrary environment variables can be preset in the context of a new task in addition to the usual program arguments.

With such a powerful system it is no longer necessary to have one PVM daemon running on every workstation in the cluster controlling only tasks on this node. Instead only one PSPVM2 daemon can be made responsible for all tasks in a ParaStation cluster: this way the whole ParaStation cluster is regarded as a multiprocessor node within the Parallel Virtual Machine. This makes it very easy for a task to find out if another task is connected to the same ParaStation cluster (see 3.1). Local tasks are able to communicate directly using the ParaStation interface. Every task that is not connected to the PSPVM2 daemon but to some other PVM daemon is regarded as remote.

2.2 Communication

With the socket based PVM every task has to maintain its own message storage of received messages *and* messages not received by the PVM application yet. Data sent through sockets has to be retrieved as soon as possible to avoid blocking the sender and fragment size is rather limited due to system constraints. A message has to be reassembled from fragments that might arrive out-of-order.

The port interface of ParaStation offers two different types of ports. Both of them distinguish messages based on the port they were sent from and a short integer tag that is assigned to each individual message by the sender. It is possible to select messages for reception or examination by source and/or tag and a user selectable compare function. The ports interface guarantees reliable in-order transfer. The implementation is much more tuned for performance than the PVM message storage. In contrast to system sockets it is not nescessary to retrieve messages from a port immediately: if resources are low the ParaStation transport layer moves messages to memory owned by the process which has created and uses a port. The process itself does not have to care for this at all.

The *stream* port grants access to incoming data as soon as it has arrived, even if the message is not yet complete. In contrast, the *datagram* port keeps incoming message fragments on hold until a message arrived completely. Therefore it is able to report the correct length of every stored message.

This powerful transport layer solves all of the problems the PVM message storage had to cope with when using sockets. It makes it possible to realize the basic idea of PSPVM2: messages sent between PSPVM2 tasks of the ParaStation cluster are directly mapped to ParaStation messages. Receive and send functions can be implemented with very few ParaStation functions. The PSPVM2 daemon and the slow message storage in the tasks are removed from the communication path. This approach is the main reason for the good performance of PSPVM2.

Every PSPVM2 task creates one ParaStation port. As this port is not only used for communication, but also as a replacement for the PVM message storage, no other way of messages into the task is possible. The socket connection to the daemon has been dropped and no direct socket connections with remote tasks are possible, although PSPVM2 tasks still support the protocol to establish such a connection. However, all they do is reject a connection request.

Messages within the ParaStation cluster are not fragmented by PSPVM2's send functions, because the ParaStation transport layer is able to transmit large messages. The messages in the PSPVM2 daemon and in the *libpvm* may still consist of fragments, but these fragments are merged into one message by the ParaStation send function. For compatibility reasons the original encoding functions and internal message structures are left unchanged, but the size of message fragments that are allocated internally during encoding of a message has been increased.

The PSPVM2 daemon is used as a message relay to remote tasks. The outgoing socket connections do not support these large packet sizes, but refragmentation of outgoing messages was already built into the original PVM daemon anyway. The other direction became more difficult: the original PVM daemon just passed incoming message fragments on to the addressed task, but now PSPVM2 tasks are no longer prepared to do their own message reassembly. This task was moved to the daemon.

2.3 (PS)PVM task IDs

PVM assigns a special 32 bit ID to every task in the virtual machine. This TID is used to identify source and destination of messages, too, so for PSPVM2 the information encoded into this TIDs had to be sufficient to reconstruct the source and destination for ParaStation messages.

PVM reserves 2 bits for status information and 12 bits for a host identifier ([2, chapter 7.1.1]). Based on this identifier tasks can recognize other tasks running on the same host, i.e. for PSPVM2 tasks running in the same cluster. The remaining 18 bits can be arbitrarily used by the PVM daemon running on the host, with only one restriction: all 18 bits zero are used to identify the daemon itself.

ParaStation uses 32 bit port identifiers for message passing. They are composed of a node number ranging from zero up to the number of machines in the cluster minus one and a port number that has to be bound to a port by the task that creates the port. This port number can be chosen by the task, but has to be unique for the node.

These two numbers are encoded more efficiently in the available 18 bits of the PSPVM2 TID. The number of bits used for the node number can be defined at compile time and defaults to eight. This yields $2^8 = 256$ nodes, which is suitable for most ParaStation clusters. The remaining 10 bits are enough to allow $2^{10} - 1 = 1023$ tasks per node [3]. It is unlikely that a normal workstation is able to support this number of processes.

The fact that the ParaStation port is part of the PSPVM2 TID has severe consequences for managing and spawning tasks. It is no longer possible to assign an arbitrary unique ID to a task as soon as it could be started. Instead the daemon has to learn which port the task has created before constructing the task's TID and making the new task known to the task that wanted to spawn it. When using a tasker the daemon has to be aware that the tasker only knows an unique serial number and not the TID of the task.

The daemon could create a port for the new task, too, but passing this port on to the new task would require an additional handshake between the daemon and the task. This would either prevent spawning several tasks in parallel or lead to a similiar construction as the one used now.

2.4 Fault Tolerance

Previously, PVM tasks were connected to their daemon through a socket. This connection served for communication *and* for fault tolerance: if the connection could not be established or was closed by either of both sides it was clear that the other side was gone.

This is not possible with ParaStation ports. One can send a message to a port that is no longer present without getting an error and there is no notification if a port is closed. Fault tolerance is guaranteed differently with PSPVM2 and ParaStation. A process can request to be sent a specific signal if a process somewhere in the ParaStation cluster dies and ask who died after receiving the signal. If such a notification is requested for a task that is not present at the moment an error is returned immediately. In addition to its port ID the daemon also propagates its process ID. Tasks use this to be killed, too, as soon as the daemon dies. The daemon requests a SIGCHLD signal for every known task and asks both the system (for a shadow PVM daemon) and the ParaStation system for dead childs in the handler for this signal.

3 Implementation

PSPVM2 is derived from the PVM V3.3.11 sources for Unix sockets. It is still possible to compile the normal PVM for sockets from the PSPVM2 sources. The differences in the source code are seperated with `#ifdef PARASTATION`. A new file `pspvm.h` is used for defintions used by both PSPVM2 daemon and the

[3] Remember, all bits zero is reserved for the local daemon.

libpvm. The core of the library is a seperate file called `lpvmpsi.c`, because the nescessary changes to the original `lpvm.c` were too severe.

New PVM architectures (`PSALPHA, PSLINUX`) have been created for those machines that are supported by ParaStation at the moment. The machine configuration for these architectures defines "`PARASTATION`". The script `pvmgetarch` has been modified to report the `PS` variante of the original architecture if the ParaStation software is installed in `/PSI` and the variable `SOCKET_PVM` is unset or empty. The `Makefile.aimk` in "`$PVM_ROOT/src`" was changed and moved to the architecture subdirectories where `aimk` will use the correct version automatically.

This section describes only the changes that were nescessary to implement the PSPVM2 concept. Please refer to [2, chapter 7, "How PVM Works"] for a more general description of PVM internals.

3.1 PSPVM2 task ID encoding

The encoding is controlled by defines in `pspvm.h`. `TID_NODE_BITS` defines the number of bits used for the node number. It can be overriden with compiler options, so during installation of PSPVM2 this encoding can be tailored to the actual number of nodes available.

Given a PSPVM2 TID of a local task the macro `DECODE_LTID(tid)` calculates the task's ParaStation port ID, whereas `ENCODE_TID(tid, port)` turns a port ID and a local TID into the PSPVM2 TID of the tasks that uses the port. The local TID is required to set the host number correctly.

3.2 Communication

3.2.1 Replacing Sockets with Ports

In the PSPVM2 daemon the socket `loclsock` has been replaced by the ParaStation *stream* port `psi_port`. A *stream* port is used because the daemon doesn't require correct message lengths. This improves performance when messages are passed through the daemon to remote tasks and reassembles the behaviour of the original `loclsock`. All task-daemon communication is done through this port. The PSPVM2 daemon still uses the Unix sockets `netsock` and `ppnetsock` for communication with other daemons.

All PSPVM2 tasks initialize `psi_port` and their own *datagram* port on startup. The defines `DECODE_TID_ME(tid,me)` and `DECODE_TID_LCL(tid,lcl)` return `psi_port` in case they recognize that the `tid` is not one of a local task.

3.2.2 Message Handling

The PSPVM2 daemon was made responsible for assembling all fragments sent from outside the ParaStation cluster to a local task. The files `pkmsg.[h|c]` implement message reassembly that takes into account that fragments may arrive out-of-order through remote connections. The daemon keeps a list (`t_txm`) of incomplete messages for every local task. As soon as a message is complete, its fragments are added in the correct order to the list `t_txq` and sent to the task as one message by `locloutput()`.

For messages sent within the cluster from task to task the overhead of additional headers can be avoided in certain circumstances. The message tag has to be suitable. In contrast to PVM ParaStation supports only tags of 6 bits instead of 32 bits. As long as the tag is smaller than $2^6 - 1 = 63$ and the encoding is raw the message can be sent without headers, because all information can be restored from the information ParaStation delivers (length, source and tag). If any of this criterions is not met PSPVM2 falls back to sending the message with header and indicates this by using the tag `PSI_MSG_TAG_PVM_MSG = 63`.

Task-daemon messages always include the usual PVM message and packet header, but several fragments are merged into one packet, too. This is nescessary because the destination can be a remote task and the header contains essential fields like the wait ID that cannot be reconstructed.

3.2.3 Sending and Receiving in Libpvm

Sending is done in the function `mroute()`. Using this name avoids some source code changes, but in contrast to the socket PVM this function does not receive any messages from ParaStation. It is only called to send a message. At first it finds out if the message can be sent without headers and which port it is going to. This is the daemon's port if it is a remote task. Then it sets up the header data if necessary and starts sending all fragments.

`pvm_psend()` is very similiar and can send data to remote tasks, too. It is more efficient than `pvm_send()` because only one chunk of memory has to be sent and of course saves the costs of encoding and setting up the message before sending. It produces exactly the same messages as `mroute()`, so `pvm_psend()` and `pvm_precv()` can be mixed with normal send/receive calls.

Receiving data consists of two steps: finding a suitable message at the ParaStation port and copying it to the own memory. The first step is done with the function `pspvm_fselect()`. It uses the custom compare function `psi_compare()` to decide which messages in the ParaStation port should be matched. It returns informations about the matched message together with a ParaStation handle that can be used to receive exactly this message.

`psi_compare()` uses global variables to get additional arguments from `pspvm_fselect()` and to store the result. The TID and tag could not be passed otherwise. When looking for a message in the port ParaStation calls `psi_compare()` with the PSI tag and source and grants access to the first bytes of the message that contain the headers, if present at all. `psi_compare()` passes this information on to `pspvm_get_msg_infos()` which fills a `struct pspvm_msg_infos` with the length, the PVM source and tag, encoding and wait ID.

In the next step `pspvm_receive()` can take the PSI message handle and the information collected by `pspvm_get_msg_infos()` to turn this message into a PVM[4] message. It simply allocates one fragment, copies the information about the message and receives all the data into this fragment. In `pvm_precv()` no PVM message is required, instead the data is received into the PVM task's buffer directly.

Some special treatment is required for receiving task control messages that are sent to the task by the daemon. If such a message is encountered in `psi_compare()`, it is matched immediately, received by whatever function wanted to get a message and handled by `pvmmctl()`. Afterwards, the receive function restarts the search for the message the PVM task was looking for.

[4] There is no special PSPVM2 message—PSPVM2 uses exactly the same message structures and functions as the original PVM.

3.2.4 Message Probing

In the original PVM messages that were not received with any PVM receive call could already be addressed with message IDs just like received messages. This is no longer possible with PSPVM2, because such unreceived messages are residing in the port and are not associated with a PVM message structure yet. Therefore a special treatment of the message ID returned by `pvm_probe()` had to be found.

`pvm_probe()` returns a message ID that must not be used for anything else but getting information about the corresponding message with `pvm_bufinfo()`, so it is sufficient to find a way to pass the information found by `pvm_probe()` on to `pvm_bufinfo()`. As it is impossible to know if a message ID created by `pvm_probe()` is ever going to be used at all or how often, allocating memory is out of question.

The solution is saving the information in a circular buffer. `pvm_probe()` keeps a counter that is incremented for every message it finds with `pspvm_fselect()`. This counter (`pspvm_probe_mids_num`) starts at `PSPVM_FIRST_BUF_ID + 1 = 0x20000001`, a value normal buffer IDs are not allowed to reach. The modulo of this counter with the size of the buffer is used as index into the buffer to find the entry the current message informations are stored in, thus erasing the oldest entries as soon as the buffer is full and new ones have to be added.

`pvm_bufinfo()` handles two special cases: `PSPVM_FIRST_BUF_ID` itself is used to identify the information of the message `psi_compare()` is currently looking at. This is nescessary for passing information to the callback function a PVM user can set with `pvm_setrcvf()`. Message IDs larger than `PSPVM_FIRST_BUF_ID` are recognized as IDs faked by `pvm_probe()` and the information is retrieved from the circular buffer. This buffer `pspvm_probe_infos[]` has `PSPVM_NUM_BUF_INFOS = 20` entries, so the result of the last 20 calls of `pvm_probe()` can be used to get the corresponding information. The information of previous calls are lost. This is recognised and the error code `PvmNoSuchBuf` is returned then.

3.2.5 Waiting for Communication

In the original PVM tasks could ask for the sockets PVM was using for communication with pvm_getfds() and add these sockets to its sockets before waiting for communication with a `select()`. This approach fails for all PVM systems that use no sockets for communication.

In PSPVM2 a new function `pvm_select()` was introduced that acts just like the system `select()`, but also returns if a PVM send/receive call might be ready now. The system call was *not* silently replaced, because a program might really want to ignore PVM communication while doing some file IO. The PSPVM2 console uses this call to wait nicely.

In PSPVM2 this function calls the `select()` first in order to be able to return valid FD sets. Furthermore, this call prevents busy waiting due to a small timeout. If the `select()` did not find any ready sockets the ParaStation port is checked for message ready for receiving, too. If it is not ready either, this process repeats until the user selected timeout is over.

3.3 Task Management

3.3.1 Contacting the Daemon

The PSPVM2 daemon publishes both its port ID and its process ID in "/tmp/pspvmd.<uid>". This information is only available on the machine where the daemon was started, so tasks that are spawned on other nodes of the ParaStation cluster get the information in the environment variable PSPVM_DAEMON. It would be possible to use a directory that is mounted on all nodes in the cluster instead of "/tmp", but this directory would have to be writeable for all users and cleaning up old files would not be done automatically.

After obtaining the information, a PSPVM2 task first requests to be sent a SIGUSR1 signal when the daemon dies and quits immediately if the daemon is already dead. The signal SIGUSR1 was chosen because a program might catch SIGTERM. The signal handler for SIGUSR1 simply prints an error message and stops the program.

After confirming that the daemon is available, a PSPVM2 task has to contact it. It does this by sending a connect message to the PSPVM2 port with a special tag and waiting for a reply. The daemon checks for this tag regularly in `loclconn()` and either accepts or rejects the task based on the protocol version in the connect message. In case of acceptance a PSPVM2 TID is constructed for the port the message came from and a new task is added to the internal list. Now that this task is known it may send `TM_CONN*` messages. The connect message also contains the PID of the connecting task, so the daemon will be notified if this task dies, too.

3.3.2 Spawning Tasks

The PSPVM2 daemon still has the function `forkexec()`, but the combination of the system calls `fork()` and `exec()` is replaced by a call to the spawn function of ParaStation. The notification mechanism of ParaStation ensures that a SIGCHLD is generated if the spawned task dies, even if it was not even able to send a connect message to the daemon.

A new task entry is added to the list, but it has no valid TID yet, therefore the spawn request cannot be answered at this moment. A reply has to be delayed until all tasks the sender asked for have either contacted the daemon and got their TID or failed. The daemon keeps track of the pending spawn request in a `struct wait_reconnect`. Every task that was spawned in response to this request has a pointer to the same instance of this structure. It contains the total number of requested copies, the number of reconnected tasks, their TIDs and the number of failed tasks that died before contacting the daemon. Every time one of these tasks reconnects or dies this structure is updated until finally the reply message can be filled with all known TIDs or failure codes and be sent back.

This startup procedure slows down spawning compared to the original PVM. Table 1 shows time required by a master to start a variable number of slaves, broadcast a message to them and receive a message from each of them. But compared to the performance gains for the communication (see section 4.2) this overhead is not very high and spawning is not a critical part of a communication library. The new version is much faster than the old PSPVM version.

Table 1: Spawn Performance

# Tasks	PVM [s]	PSPVM [s]	PSPVM2 [s]
2	0.035	0.1	0.2
4	0.055	0.6	0.5
8	0.14	2.5	1.1
16	0.22	7.0	2.2
32	1.25	11.5	6.0

Spawned tasks have their entry in the internal list created when they are spawned, so later this entry has to be found when they are reconnecting. Their process ID is not sufficient, because the `PvmTaskDebug` flag is supported and the process ID known to the PSPVM2 daemon might be the one of the debugger. Therefore the unique identifier used by the original PVM is both saved in the task's entry and passed to the task in the environment variable `PSPVM_SERNUM`. A spawned tasks looks for these variable and contacts its daemon with a slightly different connect message that also contains the real process ID and this serial number. The daemon searches its list for this number and sets the correct TID now.

3.3.3 Output Redirection

With the "fork/exec()" combination it was easy to create a pipe that connects the output file descriptors of a spawned process with a file descriptor of the daemon. This is still accomplished, but the implementation became a bit more complicated. Before spawning a task the daemon creates a Unix socket for this task (`t_listener`) and tells the ParaStation system to redirect the output of the spawned task to this port.

In order to spawn several tasks simultaneously the PSPVM2 daemon does not wait for the connection in `forkexec()`. Instead, this socket is added to the set of FDs that may become ready. In the main loop every task's listener socket is checked for the incoming connection. As soon as it is ready it is accepted in `loclstout()` and the new connection becomes `t_out`.

This initialisation is further complicated by PVM's output trace mechanism. Output messages include the TID of the task that generated the output, so accepting and sending the output messages can only be done after the task reconnected. If the task failed before reconnecting there is still a pending connection that might deliver some output. This output could even explain why the task's initialisation failed, so in this special case output messages are sent with the serial number of the task instead of a real PSPVM2 TID.

However, the PVM console believed that such a task was not spawned at all and was not prepared to get output messages for it. This was fixed by reporting the error code `PvmNoTask` in case a task failed after spawning and before reconnecting. The PSPVM2 console now knows this error code and removes the job for the spawned tasks only after receiving the output of all of them.

3.4 Necessary Changes in the Ports Interface

It was mentioned several times that the PSPVM2 approach could be realised only because of the powerful ParaStation port interface. However, some features were not present right from the start but were added later as required for PSPVM2. This is an overview of what was added and why:

Moving messages to user space: ParaStation uses shared memory of a fixed size to store messages received from the hardware. Moving messages from the shared memory to user space resolves this resource limit. Without this feature PSPVM2 would have had to implement its own message storage with all the disadvantages regarding performance and complexity. Now all other applications using the port interface profit from this improvement in robustness and simplicity, for example the MPI port for ParaStation ([4]).

Setting environment variables: Argument passing to spawned tasks is much easier now, especially if using the program arguments is prohibited. An alternative method supported by ParaStation is a port of the parent task that the spawned task can ask for, but this fails if the task contacting the ParaStation daemon is a child of the task spawned by this daemon. This would have prevented the support of spawning a task under debugger control.

Output redirection: ParaStation already redirected the *stdout* and *stderr* of a spawned process to the output of its parent. This feature was extended. Now both channels can be merged into one and sent to a socket the parent can chose.

Signals when tasks die: This takes the ParaStation cluster one step closer to one large parallel machine. It is absolutely necessary to make the connection-less port interface as fault tolerant as a socket interface.

Sending a message in fragments: A much more flexible send routine was added. The main feature used in PSPVM2 is the ability to send one message in several fragments. This did not impose too much additional work on the ParaStation library, because message reassembly was already present. It was only nescessary to give the application the possibility to tell which fragment is the last in a row.

Flexible message handling: The functionality of the custom PVM message matching function was reproduced in ParaStation. A message identifier was introduced together with a receive function using this identifier. This solves the problem of receiving exactly the message that was matched before, even if other messages arrived at the same port meanwhile.

Message length: For *datagram* ports the length of a message is available now. This is essential for an intermediate system like PSPVM2 that does hardly know which message is going to arrive. It was also useful for another ParaStation project called ParaStation Active Messages ([5]).

Table 2: Performance of the protocol hierachy

Protocol layer	Alpha 21164, 500MHz Latency [μs]	Band-width [MB/s]	Pentium, 120 MHz Latency [μs]	Band-width [MB/s]
System sockets	112.2	1.1	413.6	1.0
PVM	266.7	1.0	800.0	1.0
ParaStation sockets	9.2	9.6	20.0	10.2
PSPVM	62.7	8.8	225.9	7.2
ParaStation *datagram* port	9.3	9.4	22.7	9.8
PSPVM2 `pvm_psend/precv()`	11.5	9.4	42.5	9.2
`pvm_send/recv()`	16.3	9.4	78.9	9.6

4 Tests and Benchmarks

4.1 Functionality Tests

All PVM examples (`[g]examples, tasker, hoster`) can been compiled without changes and work with PSPVM2. PSPVM2 also passed all tests of the PVM test suite ([3]) after fixing some problem the test suite had with the Alpha architecture.

PSPVM2 has passed the test suite tests in combination with another PVM daemon, too. The master server was running on a SUN4MP architecture and the slave server was PSPVM2 on Alpha. Only some tests were done with the PSPVM2 daemon as master, because `rsh` did not work from Alpha OSF 4.0 to Solaris 2.6 and therefore the slave daemon had to be started manually for every test.

4.2 Performance Tests

Message passing tests were done with two tasks on different machines. The first cluster constisted of two DEC Alpha 21164, 500MHz running OSF V4.0B and the second one of two Pentium 120 with Linux.

Latency was measured with a pingpong test and the result is one half of the round-trip time. For the bandwith an exchange test was used with two tasks sending simultaniously and receiving afterwards. All tests were repeated with message sizes ranging from 1 byte to 64KB with all powers of two.

Table 2 gives an overview of the performance on the two platforms with different transport layers that are reached when using the optimal message size for each test and configuration.

The figures plot the whole results with logarithmic scales on both axes. Each figure shows the results of one test on one machine architecture.

The performance tests show that PSPVM2 adds very little latency to the ParaStation system. On the Alpha architecture the additional latency are 2.2μs which are only 23%, resp. 19.8μs and 87% for the Pentium. This is much better than the additional overhead of PSPVM which amounts to 580% on Alpha.

Figure 1: Linux, pingpong

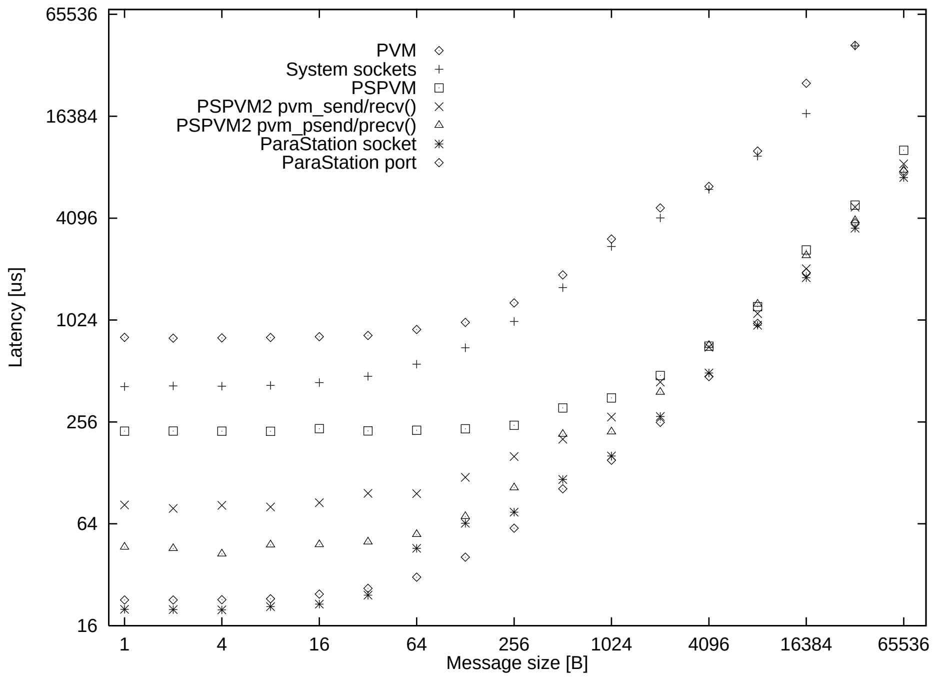


Figure 2: Alpha, pingpong

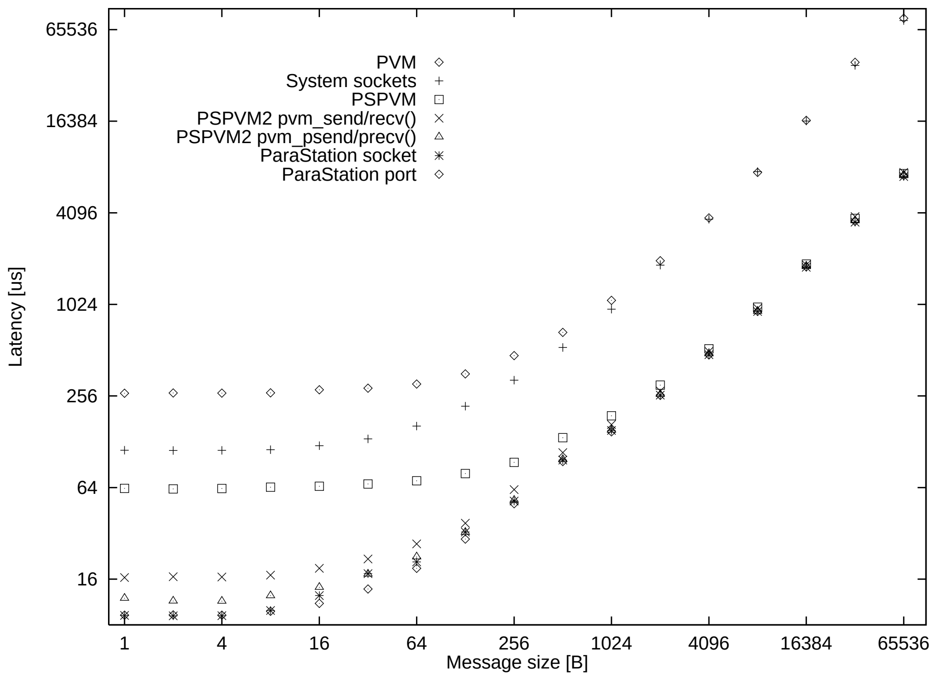


Figure 3: Linux, exchange

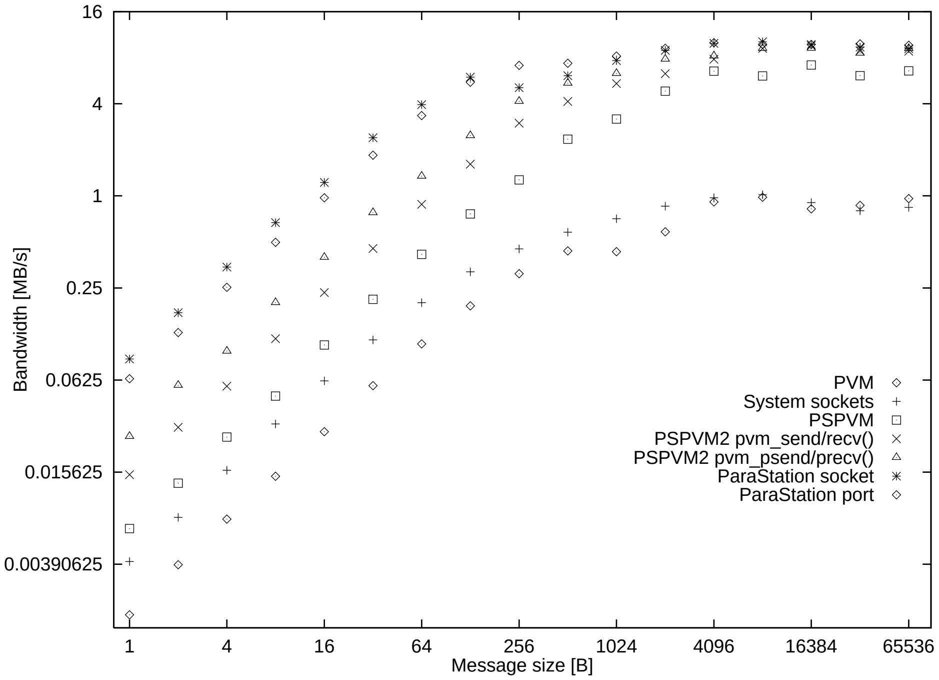
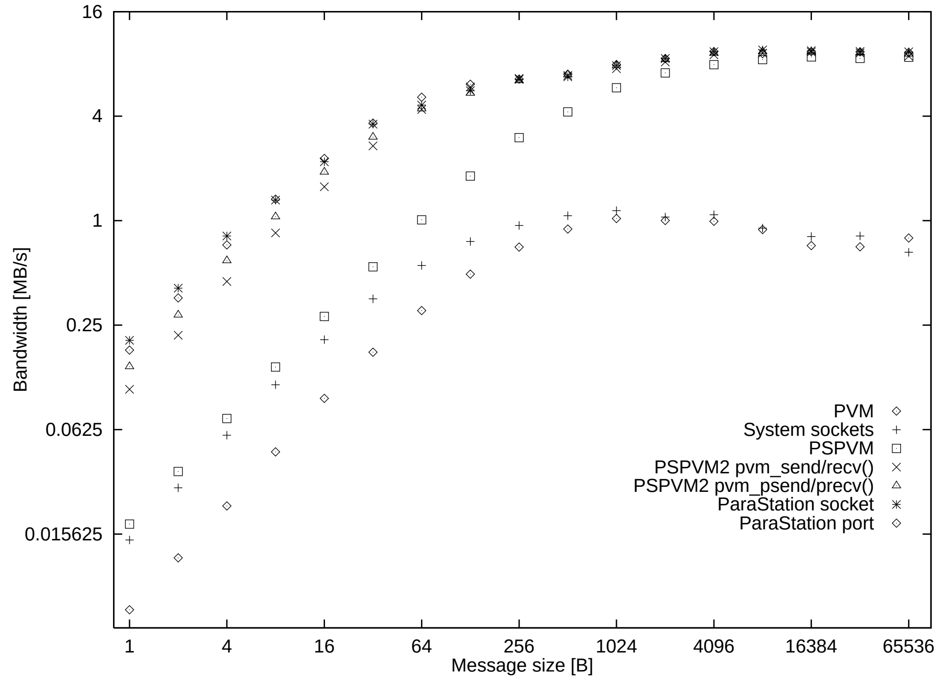


Figure 4: Alpha, exchange



As far as latency is concerned, PSPVM2 is now 5.4 times faster than PSPVM and 23 times faster than the system sockets PVM on Alpha. These factors are similiar for Pentium with factors of 5.3 and 18.8.

Especially on Alpha the normal send/receive functions of PSPVM2 perform very well compared to the `pvm_psend/precv()` functions that are usually intended to be optimized for native message passing.

The maximum bandwith of PSPVM2 that is reached with messages of 4KB has hardly increased compared to PSPVM. However, with smaller message sizes it reaches a much better bandwith: 3MB/s are already possible with messages of only 32 bytes. PSPVM reaches this value only with messages that are at least 256 bytes long. For 32 byte messages PSPVM2 is 5.6 times faster than PSPVM.

5 Related Work

Several other special PVM versions for message passing systems use an approach similiar to PSPVM2's with one daemon for the whole cluster and utilizing native message passing functions to implement PVM functions. However, spawning tasks on iPSC860, CM-5 and Paragon is much more restricted compared to ParaStation (see [2, pages 126–130]). In addition, these implementations do not allow mixing of `pvm_psend()` with `pvm_receive()`.

PVMe [6] for the IBM 9076 SP2 dropped the support for a heterogenous configuration of the Parallel Virtual Machine and does not use any of the original source. This makes it more difficult to maintain source code compatibility with the original PVM and impossible to use a PVMe-SP2 cluster together with other computers.

6 Conclusions and Future Plans

PSPVM2 has successfully reached its aims: message passing performance is now very close to the underlying transport layer and compatibility was not restricted in order to achieve this.

However, group communication has not been optimized at all. It benefits from the faster communication, but is not tailored to the ParaStation system yet. With broadcast support in the ParaStation port interface performance probably could be improved.

PSPVM2 did not change the message packing, either. If it proves to be a new bottleneck a more efficient memory handling might help. It could keep a list of free chunks with the size that is used most frequently. The PVM message fragment size is a very likely candidate for such a system. However, not only PSPVM2 but also every other PVM version would profit from performance improvements here.

ParaStation is currently being ported to the Myrinet [7] communication fabric. As soon as this port is completed successfully PSPVM2 will work with this hardware, too. In addition, adapting the PSPVM2 concept to other message passing systems should be possible.

A User Manual

PSPVM2 is very similiar to the original PVM. You can install and use it as described in the PVM guide ([2]). There are only some minor differences that are described here:

A.1 PVM or PSPVM2

You can use the same source to install and use both the ParaStation PVM and the original socket based PVM on the same machine. If you don't set the environment variable `PVM_ARCH` the PVM scripts default to install and use PSPVM2 if you have ParaStation installed. You can override this choice by setting the environment variable `SOCKET_PVM` to any value.

A.2 One PVM Daemon

PSPVM2 will only start one PVM daemon that is responsible for the whole ParaStation cluster. You do not have to add the other nodes, this daemon will spawn new processes on them automatically.

The daemon will write a log file into "`/tmp/pspvml.<uid>`" and create a file "`/tmp/pspvmd.<uid>`" for PVM programs that are started manually on the machine the daemon is running on. PVM programs started in a shell on other machines in the ParaStation cluster are *not* able to contact this daemon!

The prefix "`ps`" has been added to these filenames to make it easier to use both the normal socket PVM and PSPVM2 on the same machine. This way they can even run simultaniously and clients will not try to contact the wrong daemon if these files are not removed properly.

A.3 Waiting for Communication

The original socket PVM offers the possibility to wait for communication nicely by asking for the sockets PVM has allocated with `pvm_getfds()`. These sockets can be added to a set of sockets an application is waiting for with `select()`.

PSPVM2 does not use any sockets for PVM communication, so `pvm_getfds()` will give the correct answer that no sockets have been allocated. As a substitute a new PSPVM2 specific function has been introduced: `pvm_select()` takes the same arguments as the system call, but returns if a PSPVM2 message arrives, too.

This is the only change in the source code that has to be done in order to adapt PVM applications to PSPVM2. Programs that do not use `pvm_getfds()` do not have to be changed.

A.4 `pvm_psend/recv()`

The functions `pvm_psend()` and `pvm_precv()` may be mixed with the normal send/receive functions, i.e. they can be used to send to any task and can receive any message, respectively.

They are a bit faster than the normal message passing functions (see 4.2), but their main advantage is that they avoid message encoding and decoding. The other functions already make full use of the ParaStation communication, so in contrast to other PVM architectures you do not have to use `pvm_psend/recv()` to get fast communication.

A.5 Output Redirection

Output redirection is supported just like before, but there is a slight difference. A spawned tasks may fail before it could contact its PVM daemon. `pvm_spawn()` will report this as a total failure with the special error code `PvmNoTask`, but this task produces output, though. If an application asks for the output of spawned tasks it will get some messages it might not expect in this special case. The PVM console has been adapted to handle this case properly. Please refer to the source file "`console/cmds.c`" for more information.

B Exact Performance Results

Table 3: Linux, pingpong

Message size [B]	ParaStation port [μs]	ParaStation sockets [μs]	System sockets [μs]	Socket (PS)PVM ParaStation [μs]	Socket (PS)PVM System [μs]	PSPVM2 `pvm_psend()` [μs]	PSPVM2 normal [μs]
1	22.700	19.987	413.654	225.871	808.645	46.644	82.714
2	22.720	19.933	417.489	225.975	799.566	45.751	78.911
4	22.799	19.861	416.572	225.946	803.132	42.516	82.290
8	23.107	20.732	420.598	225.229	806.793	48.051	80.436
16	24.600	21.474	436.774	233.699	815.815	48.246	85.313
32	26.590	24.228	475.946	226.684	830.025	50.078	96.941
64	31.003	45.854	561.245	228.632	901.083	55.390	96.455
128	40.691	64.560	702.062	233.147	990.834	70.799	120.367
256	60.322	74.943	1004.460	244.535	1291.818	104.637	159.967
512	102.993	116.942	1592.569	309.833	1893.389	216.411	202.350
1024	152.074	161.080	2784.071	354.591	3085.291	223.760	273.791
2048	254.618	275.466	4105.375	481.690	4706.054	383.400	439.998
4096	474.295	498.243	6059.956	717.683	6304.217	725.333	706.923
8192	979.490	953.864	9519.577	1228.111	10194.279	1271.208	1116.663
16384	1950.122	1819.917	17000.610	2656.437	25647.089	2463.104	2058.212
32768	3843.728	3570.468	42672.058	4887.488	42941.723	3960.044	4756.428
65536	7736.487	7116.142	72868.507	10302.083	70958.984	7860.711	8562.672

Table 4: Alpha, pingpong

Message size [B]	ParaStation port [μs]	ParaStation sockets [μs]	System sockets [μs]	Socket (PS)PVM ParaStation [μs]	Socket (PS)PVM System [μs]	PSPVM2 `pvm_psend()` [μs]	PSPVM2 normal [μs]
1	9.273	9.232	112.639	63.152	266.690	11.991	16.386
2	9.312	9.170	112.200	62.670	267.694	11.484	16.581
4	9.263	9.190	112.326	63.084	267.080	11.452	16.508
8	9.829	9.959	113.420	64.285	268.398	12.450	16.979
16	11.074	12.485	120.496	65.286	281.066	14.131	18.834
32	13.770	17.384	133.701	67.540	287.898	17.257	21.703
64	18.836	20.746	162.151	70.888	306.832	22.430	27.240
128	29.322	32.651	219.110	78.985	357.564	32.499	37.277
256	50.093	52.097	325.327	93.631	471.570	52.775	62.069
512	94.761	96.847	533.115	135.998	669.890	97.897	108.492
1024	148.775	151.748	952.196	189.736	1088.142	152.640	165.820
2048	258.463	259.174	1855.173	301.542	1982.257	261.645	275.771
4096	476.425	477.796	3714.542	523.055	3788.532	484.546	502.687
8192	919.419	919.374	7643.960	982.157	7555.856	923.965	965.551
16384	1806.930	1794.290	16443.880	1886.336	16590.289	1827.406	1863.498
32768	3600.293	3554.734	38117.119	3756.783	39975.601	3634.384	3865.909
65536	7203.465	7085.702	75116.631	7413.615	77837.982	7282.159	7546.340

Table 5: Linux, exchange

Message size [B]	ParaStation port [MB]	ParaStation sockets [MB]	System sockets [MB]	Socket (PS)PVM ParaStation [MB]	Socket (PS)PVM System [MB]	PSPVM2 `pvm_psend()` [MB]	PSPVM2 normal [MB]
1	0.0637744	0.0858861	0.0040730	0.0066870	0.0018256	0.0267516	0.0150358
2	0.1277797	0.1724680	0.0079061	0.0132269	0.0038754	0.0578391	0.0306732
4	0.2523179	0.3423957	0.0160861	0.0265202	0.0077072	0.0967273	0.0569489
8	0.4965330	0.6679299	0.0322897	0.0490841	0.0146887	0.2006357	0.1165324
16	0.9705452	1.2227775	0.0617314	0.1060444	0.0287374	0.3967330	0.2332246
32	1.8423946	2.3971554	0.1143985	0.2104919	0.0574426	0.7768309	0.4526366
64	3.3465646	3.9466705	0.2002940	0.4143371	0.1078083	1.3452937	0.8795952
128	5.5284379	5.9699964	0.3185232	0.7614182	0.1909789	2.4763373	1.6101346
256	7.1224694	5.0932547	0.4500552	1.2723784	0.3095275	4.1528029	2.9858143
512	7.3531732	6.1048142	0.5787939	2.3454082	0.4367222	5.4472520	4.1371681
1024	8.1992489	7.6569319	0.7086110	3.1786711	0.4313473	6.3149798	5.4172078
2048	9.2062527	8.9438037	0.8557323	4.8333879	0.5819890	7.8310558	6.2934852
4096	10.0200180	9.9141923	0.9718778	6.5317238	0.9128802	8.2100622	7.8091484
8192	9.7771316	10.1554654	1.0170704	6.0823915	0.9793214	9.2176737	9.1874794
16384	9.7562083	9.7674016	0.9035576	7.1699853	0.8218084	9.2215925	9.5851331
32768	9.8327927	9.3691172	0.8001206	6.1056328	0.8653409	8.5606776	8.8630714
65536	9.6435709	9.1455338	0.8412586	6.5570549	0.9597529	9.1823868	8.8147160

Table 6: Alpha, exchange

Message size [B]	ParaStation port [MB]	ParaStation sockets [MB]	System sockets [MB]	Socket (PS)PVM ParaStation [MB]	Socket (PS)PVM System [MB]	PSPVM2 `pvm_psend()` [MB]	PSPVM2 normal [MB]
1	0.1794394	0.2042257	0.0144452	0.0178452	0.0057234	0.1441470	0.1066626
2	0.3589674	0.4082312	0.0288904	0.0358439	0.0113914	0.2857633	0.2186748
4	0.7249192	0.8163332	0.0580019	0.0724151	0.0227033	0.5877513	0.4456725
8	1.3296564	1.3133580	0.1132464	0.1434761	0.0465224	1.0527645	0.8500221
16	2.2799921	2.1845032	0.2061834	0.2802518	0.0945586	1.9055604	1.5686152
32	3.6451253	3.5984814	0.3546099	0.5414361	0.1745482	3.0285594	2.6932217
64	5.1407537	4.6340358	0.5514617	1.0102975	0.3033750	4.4056987	4.3719190
128	6.1017605	5.6136943	0.7592258	1.8061586	0.4915620	5.4261015	5.8977697
256	6.5406955	6.5170833	0.9390250	3.0071741	0.7045137	6.4007810	6.5963604
512	6.9773932	6.7420624	1.0685955	4.2319031	0.8948934	6.9239792	6.8695766
1024	7.9308089	7.8297036	1.1422371	5.8362265	1.0295564	7.8074631	7.5081065
2048	8.6003100	8.5998389	1.0487604	7.0979012	1.0017569	8.4732043	8.2130018
4096	9.4184162	9.4140145	1.0814780	7.9233221	0.9913451	9.2250307	9.0365072
8192	9.1742115	9.6300336	0.9055576	8.4667435	0.8884948	9.4167262	9.2173105
16384	9.4276387	9.5440483	0.8105767	8.7800968	0.7187375	9.3662037	9.3758809
32768	9.3981902	9.4386147	0.8154832	8.6058754	0.7063486	9.2540835	9.2343580
65536	9.3303744	9.3897788	0.6577822	8.7469956	0.7937325	9.1153197	8.8557487

References

[1] Thomas M. Warschko, Joachim M. Blum, and Walter F. Tichy. The ParaStation Project: Using Workstations as Building Blocks for Parallel Computing. In *Proceedings of the International Conference on Parallel and Distributed Processing, Techniques and Applications (PDPTA'96), New Horizons*, pages 375–386, Sunnyvale, California, USA, August 9–11, 1996.

[2] A. Beguelin, J. Dongarra, Al Geist, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 User's Guide and Reference Manual.* ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[3] H. Casanova, J. Dongarra, P. J. Muci. *A Test Suite for PVM*, March, 1995, `http://www.epm.ornl.gov/pvm/tester.html`

[4] Patrick Ohly, Philipp Mergenthaler. *MPICH for ParaStation*

[5] ParaStation Active Messages

[6] M. Bernaschi, G. Richelli. *PVMe: an enhanced implementation of PVM for the IBM 9076 SP2.* ISCA 92

[7] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, Wen-King Su. *Myrinet: A Gigabit-per-Second Local Area Network* In *IEEE Micro, volume 15, 1 Feb 1995*, pages 29–36.