Experience with a Portability Layer for Implementing Parallel Programming Systems

Tim Rühl Henri Bal
Raoul Bhoedjang Koen Langendoen
Dept. of Mathematics and Computer Science
Vrije Universiteit, Amsterdam

Gregory Benson
Dept. of Computer Science
University of California, Davis

Abstract

Panda is a virtual machine designed to support portable implementations of parallel programming systems. It provides communication primitives and thread support to higher-level layers (such as a runtime system). We have used Panda to implement four parallel programming systems: Orca, data parallel Orca, PVM, and SR. The paper describes our experiences in implementing these systems using Panda and it evaluates the performance of the Panda-based implementations.

1 Introduction

Portability is one of the most important issues in designing parallel software. The portability of parallel applications can be enhanced by using portable programming systems, but this leaves many of the problems to the implementor of such systems. In particular, it is difficult to obtain both portability and efficiency. In our research on the Orca [2] programming system, we use the well-known implementation technique of a virtual machine to achieve portability. We have designed a virtual machine, called *Panda* [4], that hides machine specific details from its users. Panda provides support for threads and communication to higher software layers (such as the Orca runtime system), in a machine and operating system independent way. To obtain high efficiency, Panda is designed as a flexible system consisting of building blocks that can be configured statically (i.e., when the system is compiled). For example, if the underlying machine provides reliable communication, Panda can be configured to make use of that; if the communication is unreliable, Panda can be configured to use its own reliability protocols. Either way, the primitive provided to higher software layers is the same.

Panda was originally designed as an efficient portability layer for implementing the Orca runtime system (RTS). During the course of our research, we also used the Panda layer to implement several other parallel programming systems. Panda has many advantages to the implementors of such systems. It provides several modules (for threads, synchronization, and communication) that are useful for many systems. These modules have well-defined interfaces, and have been implemented, tested, and optimized on many machines, so reusing them for other programming systems will save development time. In addition, we have experienced that using Panda often results in more modular systems. A potential disadvantage of using Panda is the overhead of its layered approach. We address this problem by using simple interfaces between Panda and the other layers and by using static configuration techniques.

In general, the Panda system can be used for implementing parallel programming systems that exploit medium-grained or coarse-grained parallelism and that execute on homogeneous distributed-memory machines (such as multicomputers or collections of workstations). In this paper, we describe our experiences using Panda to implement four programming systems:

- the Orca parallel language, for which the Panda system was originally designed [4],
- a system based on Orca, but with extensions to support both task and data parallelism in an integrated way [9],
- the PVM message passing system [15], and
- the SR concurrent programming language [1].

The outline of the paper is as follows. In Section 2 we first look at other approaches for portable implementations of programming systems. In Section 3 we outline the design of the Panda system. In Section 4 we discuss our experiences in using Panda to implement the four programming systems and in Section 5 we analyze the performance of these systems. Finally, in Section 6, we present our conclusions.

2 Related Work

Several other systems exist that, like Panda, support portable implementations of different programming environments. We discuss several approaches below. Other related efforts include portable communication interfaces (e.g., PVM [15] and MPI [6]) and portable threads packages (e.g., Pthreads [12], Chant [8], and uThread [14]).

Converse [11] is a framework that allows interoperability between different programming systems. Using Converse, it is possible to write different modules of a parallel program in different languages, and have these cooperate. In this sense, Converse is much more ambitious than Panda, which is intended for implementing different languages but not necessarily for interoperability between these implementations. Also, Converse provides functionality (e.g., load balancers) that in our approach would be implemented in the language-dependent RTS. Both Converse and Panda provide point-to-point and multicast communication, although in Panda the semantics of multicast are stronger. Panda supports totally-ordered multicast [10], which is useful for implementing data replication techniques that are used by Distributed Shared Memory systems.

Nexus [7] is a portable library that has been used to implement a range of programming systems, such as Compositional C++, Fortran-M, MPI, and others. An important goal in Nexus is to provide flexible mechanisms for threads and communication. For example, it should be possible to deliver an incoming message to an existing thread or to create a new thread for servicing the message. The key idea to obtain this flexibility is to decouple the specification of a communication endpoint from the specification of the receiver thread. Nexus provides two basic mechanisms for this: a *global pointer* denotes an endpoint (an address within a given address space); a *remote service request* asynchronously invokes a handler function within an address space denoted by a global pointer. These two low-level primitives can be used to implement higher-level abstractions, such as message passing or Remote Procedure Calls.

PORTS [5] is perhaps the most ambitious undertaking to obtain a portable common runtime system. A consortium of several researchers is currently working on the specification of PORTS. The PORTS interface supports threads (based on a subset of Pthreads [12])

Table 1: Overview of the Panda interface

Threads	create, exit, join, yield, self, set priority
Synchronization	
mutex	lock, trylock, unlock
condition variable	wait, timed wait, signal, broadcast
Communication	
Message Passing	register, send, receive, poll
Remote Procedure Call	register, call, reply
Group Communication	join, leave, send

and communication. Communication support is given for shared memory machines, message passing (based on a subset of MPI), and network DMA. Other functions in the PORTS interface support timers, tracing, and processor allocation. Since the PORTS interface is still under development, a comparison with our work on Panda would be premature.

Horus [16] is a system that provides group communication to support distributed applications. It uses a number of "microprotocols" with standardized interfaces, that can be stacked on each other (much like Lego blocks) to form a group protocol. The system is very flexible, and even allows dynamic configuration of protocol stacks. There are some similarities between Horus and Panda. For example, with both systems the protocols can be tuned depending on the functionality of the target system. Panda, however, is much more oriented to parallel programming and runtime systems, while Horus is a general system that also supports other kinds of distributed applications (e.g., fault-tolerant applications, multi-media applications). An important difference is that Panda is configured entirely during compile-time.

3 Panda

In this section we give an overview of the Panda system. We first discuss the interface provided by Panda to higher-level layers (language-specific runtime systems). Next, we outline the internal structure of Panda, and explain the techniques we use for making Panda both portable and efficient.

3.1 The Panda Interface

The Panda interface consists of primitives (procedures) that give support for threads, synchronization, and communication. An overview of the Panda interface is given in Table 1. As we will discuss later, the interface can be extended by the implementor of a programming system.

The support for threads is roughly based on the Pthreads interface. On most systems, Panda supports preemptively scheduled, prioritized threads. If preemption or priorities cannot be implemented on the underlying operating system, the runtime system or compiler can use calls to a *yield* primitive, which causes a thread switch. As an example, a Distributed Shared Memory system like Orca should always service incoming requests for remote memory operations in a finite time. For such systems, it is important to be able to preempt (possibly long-running) computations. On operating systems that do not support preemptive scheduling, the Orca compiler therefore uses calls to the *yield* primitive to force preemption at certain points in the program.

Two kinds of primitives are supported for synchronization between threads: mutexes and condition variables. A time-out can be given to the *wait* primitive on a condition variable, which is used to implement reliable communication protocols on unreliable networks.

Communication in Panda is primarily based on the upcall model. The arrival of a message causes a function to be invoked in the destination process; this function is defined by the user (e.g., the RTS) and must be registered beforehand. Conceptually, the upcall function is handled by a new thread of control, but the implementation typically uses daemon threads for handling messages. To allow an efficient and deadlock-free implementation with daemon threads, the upcall function is not allowed to block on a condition that depends on the arrival of new messages [3].

Three types of reliable communication are provided: point-to-point message passing (MP), Remote Procedure Call (RPC), and group communication. With message passing, the sender can either wait until the message has been delivered at the destination processor (synchronous MP) or it can continue immediately (asynchronous MP). Messages can be received using an upcall function or with an explicit receive call. With the RPC model, a client can invoke a procedure within a remote process (the server) and wait until the server has returned a result. So, RPC is 2-way communication, whereas MP is 1-way.

Group communication in Panda provides totally-ordered multicast, which guarantees that all multicast messages arrive in the same order at all destinations [10]. If two processes simultaneously multicast a message, either all receivers get the first message first or they all get the second message first. These strong semantics are useful for implementing replicated shared data, because the replicas can be kept consistent by multicasting the updates using a total ordering [2]. Panda's group communication primitives allow processes to join (or leave) a given group and to send a message to a group. Group messages are received using an upcall function that is specified in the *join* primitive.

3.2 Structure of the Panda Portability Layer

Above, we described the interface that Panda provides to a runtime system. To implement a new language on top of Panda, this interface is basically all that is needed. To understand the performance and portability of Panda, however, it is also useful to study the internal structure of the Panda system. Below, we give an overview of this structure.

The main difficulty in making Panda both portable and efficient is how to make the best use of the primitives provided by the underlying platform (i.e., the operating system and hardware). Panda should run on a variety of platforms that differ widely in their communication support. We have identified three important areas in which these systems differ:

- The low-level communication primitives may either be reliable or unreliable. For example, the TCP protocol and the send/receive primitives of most multicomputers are reliable; the UDP/IP protocol is unreliable.
- The underlying system may or may not provide totally-ordered multicast.
- Some systems support arbitrarily-sized messages, but others impose an upper bound on message size (so fragmentation and reassembly are required for larger messages).

A naive way to implement Panda would be to always assume the worst case (unreliable communication, no total ordering, and messages with a limited size), but this would be inefficient on many systems. Our solution to this problem is as follows. We structure the Panda

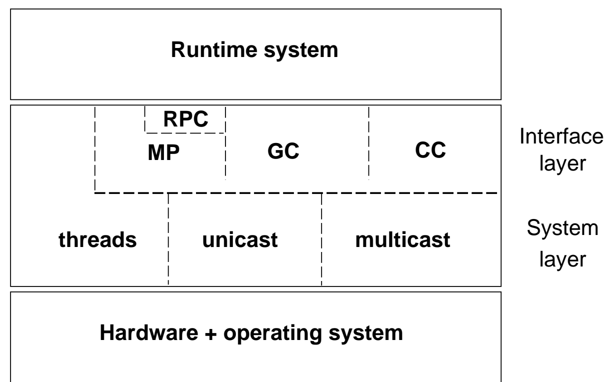

Figure 1: Structure of the Panda system. The system layer provides low-level primitives. The interface layer contains modules for Message Passing (MP), Remote Procedure Call (RPC), Group Communication (GC), and Collective Communication (CC).

system using a number of *modules*. Each module has a well-defined interface, but it can have multiple implementations. The modules belong to one of two layers, as shown in Figure 1:

- The *system layer* provides threads and low-level communication primitives. The communication primitives are implemented on top of the underlying platform.
- The *interface layer* contains higher-level communication modules, such as message passing, RPC, and group communication. The modules in the interface layer are implemented on top of the system layer.

The interface of the system layer primitives is fixed, so the signatures (number and types of parameters) of all procedures are identical on all target platforms. The *semantics* of the primitives, however, change from platform to platform. For each platform, the system layer primitives may or may not be reliable or totally-ordered, and may or may not accept messages of arbitrary size. The choices are expressed in a set of *platform parameters*.

For each module in the interface layer, we have multiple implementations, depending on the semantics of the system layer primitives. For example, if the *unicast* primitive of the system layer is unreliable, the MP module will be implemented using a time-out and retransmission protocol; if unicast is reliable, the MP module will be straightforward and not have this overhead. For the group communication module we also have multiple implementations, depending on the need for retransmissions, ordering, and fragmentation. So, conceptually we have one implementation of each interface layer module for every combination of the platform parameters. (In practice, we use conditional compilation to reduce the number of source files.)

The threads module is placed in the system layer, so it can access the operating system's thread primitives, if available. On Solaris, for example, the threads module merely wraps up the primitives of the Solaris threads library to make them look like Panda threads to the interface layer and runtime system. On systems without (efficient) thread support, we use a user-level threads package.

The configuration of the Panda system is done at compile-time, using the platform parameters described above. The structure of the system is flexible and extendible. For example, users can add new modules to the interface layer. An example is the Collective Communications module in Figure 1, which we added for implementing data parallel extensions to Orca. An interface layer module can also be built on top of other modules in this layer. An

example is the RPC module, which is built on top of (reliable) MP. Finally, and perhaps most importantly, the Panda system is highly portable. All platform-dependent parts are isolated in the system layer. Implementing Panda on a new target platform thus comes down to: (1) implementing (or porting) the system layer and (2) configuring the interface layer. Panda has been implemented on several operating systems (Amoeba, SunOS, Solaris, and Parix) and machines (the CM-5, Parsytec GCel, Parsytec PowerXplorer, CS-2, and a Myrinet cluster); ports to other systems (including the SP2) currently are under way.

4 Implementing Parallel Programming Systems on Panda

We will now describe our experiences in implementing four different programming systems on top of Panda.

4.1 Orca and Data Parallel Orca

Orca is a task-parallel language for writing parallel applications that run on distributed memory systems. Its programming model is a form of object-based Distributed Shared Memory [2]. Objects in Orca are variables of abstract data types that can be shared by different processes. A recent extension to Orca [9] also supports data parallel programming, by allowing objects to be partitioned among multiple machines. Below, we first discuss the implementation of the original Orca language on Panda; after that, we discuss the data parallel extension.

The initial implementation of Orca [2] ran directly on top of the Amoeba distributed operating system, so it was not portable. The Panda system was designed as a cornerstone of a portable implementation of Orca. Orca is implemented on top of Panda as follows. An Orca process is implemented as a Panda thread, so all Orca processes on the same machine run in the same address space. An Orca object is stored either on one machine or it is replicated on all machines that can access it. The Orca compiler and runtime system together decide (transparent to the programmer) which objects will be replicated. In general, objects that are read frequently and modified infrequently are replicated, to decrease communication overhead.

An operation on a nonreplicated remote object is implemented using a Panda Remote Procedure Call, which requests the RTS on the remote machine to execute the operation. One complication is that operations in Orca may block on a guard condition. Since upcall functions in Panda are not allowed to block arbitrarily, we create a *continuation* in this case, containing the operation and its parameters. Whenever the object is changed, its continuations are picked up and the operations are tried again. An alternative solution (which we originally used) is to create a new thread of control for the upcall handler, but this has a high context switching and memory overhead [3].

For operations on replicated objects, we use an entirely different strategy. If the operation does not modify the object, it is executed locally, without any communication. If, however, the operation does modify the object, the operation and its parameters are sent to all machines containing a replica, using Panda's totally-ordered multicast. All these machines execute the operation on their local copy, thus updating the replica. The total ordering guarantees that all machines receive all updates in the same order, so all replicas will remain consistent [2].

The data parallel extension of Orca is based on partitioned objects, which are objects containing arrays that are partitioned among multiple processors. Operations on these partitioned objects conceptually update all elements at the same time. This allows the runtime system to execute the operation in parallel on all machines that own a partition.

When a data parallel operation on a partitioned object is invoked, the operation and its arguments are broadcast using totally-ordered group communication. If a processor needs to read values stored on another machine, these values are either fetched on demand or they are prefetched before the operation begins. After the operation has finished, all results are combined (using a gather or reduction operation) and the final result is returned to the invoker of the operation.

To implement data parallel operations efficiently using Panda, we have added a collective communication module to the interface layer (see Figure 1). This module provides the primitives to gather or reduce data, and to perform barrier synchronizations. The RTS for data parallel Orca uses this module in combination with the existing group communication module to implemented partitioned objects.

4.2 PVM

PVM is a popular message-passing system with a large number of users. PVM allows Unix processes to communicate via point-to-point and group communication and takes care of data conversions between different architectures. We ported a subset of the PVM Solaris implementation (version 3.3.8) to Panda. We implemented only PVM's core message passing functionality. The Panda-based PVM system therefore only runs on homogeneous systems and it requires a few changes to the initialization code of PVM programs.

By default, the original PVM implementation routes all communication through separate *daemon processes*: application processes pass their data to a local daemon process (using Unix domain sockets) which then transmits the data to its destination (using UDP). The receiving daemon process delivers the data to the application process. This strategy results in expensive process switches and copying overhead. By implementing *all* functionality in a library rather than using an additional process, our implementation avoids these overheads. The recent "direct routing" extensions of PVM support a similar efficient scheme (using TCP).

PVM applications issue a (blocking or nonblocking) call to receive messages. In these explicit receive calls, the user can specify both the sender and the *tag* of the message it wants to receive. Although Panda's Message Passing module includes a blocking receive call, this call does not allow its caller to specify a tag. Therefore, we chose to receive all messages asynchronously via upcalls. Each upcall puts the message just received in a queue. The PVM receive call checks this queue for messages that match the specified sender and message tag.

Implementing PVM's message construction primitives using Panda's message interface was a little troublesome. PVM's pack and unpack functions treat message buffers as FIFO queues, so the sender and receiver must add and remove data in the same order. Panda messages, on the other hand, behave like stacks. Since Panda allows us to obtain a pointer to data in a message without copying the data out of the message, we were able to implement PVM's FIFO model on Panda's stack model without extra copying. The receive upcall builds an array of pointers into the received message; this array is traversed in reverse order as each data item is unpacked.

Another complication with PVM's send primitives is that the programmer is allowed to reuse the buffer immediately after the send returns. In Panda, however, the Message Passing module requires the programmer to wait for an acknowledgement before the buffer can be reused. We resolve this difference by optimistically letting the program continue. Usually, the send buffer will be reinitialized using another Panda message, but sometimes another operation may be invoked on the same message. Only in that exceptional case, we block the invoker until the acknowledgement arrives.

4.3 SR

SR [1] is a task-parallel language for writing concurrent programs that run on both shared memory multiprocessors and distributed memory systems. The main language elements are operations, processes, resources, and virtual machines. Operations represent communication channels through which processes communicate. Resources are dynamic modules which contain processes and variables. A resource is created on a virtual machine, or VM, which represents a single address space. Virtual machines can reside on different physical machines. Through orthogonal use of SR's basic communication primitives, a programmer can employ asynchronous message passing, synchronous message passing, process creation, Remote Procedure Call, and rendezvous.

The original implementation of SR runs on top of Unix operating systems that support socket-based communication (TCP). A custom user-level threads package is used to simulate concurrency on uniprocessors. A multiprocessor version of SR assigns a job server to each processor; each job server schedules user-level threads from a global ready-queue. In order to support distributed virtual machines an execution manager process, *srx*, coordinates the creation and destruction of virtual machines, and global deadlock detection.

For the Panda implementation of SR we replaced the custom user-level threads package with Panda threads. SR's internal synchronization mechanisms were replaced with Panda's mutexes and condition variables. The TCP-based communication routines were replaced with message passing functions from the Panda MP module. The *srx* process was modified to run as a thread in the main VM.

In SR, VMs can be dynamically created at run time. In Panda, however, the number of nodes used for a program must be specified at program initiation. Therefore, in the Panda implementation of SR, a program can only create as many VMs as there are Panda nodes. If a VM is destroyed, the node on which it was running can be reused for another VM.

When an SR program invokes an operation, the RTS creates an "invocation block," which is simply a region of memory. If the operation is serviced on a remote VM, the invocation block is transferred via the "write" system call through a TCP socket. The Panda MP module, however, operates on Panda messages. In order to avoid extra copying in the Panda implementation of SR, all invocation blocks are created as Panda messages.

5 Performance

The Panda system makes it easier to build portable implementations of parallel languages. Since Panda adds extra layers of software, this approach may have a certain runtime overhead. On the other hand, the Panda modules have been optimized carefully, which may compensate this potential disadvantage. In this section, we discuss these performance issues. We do so by studying how efficient the Panda-based implementations are compared to the original implementations running on the same platform. For data parallel Orca, however, only a Panda-based system exists. Since we cannot compare it against another system, we do not discuss the performance of this system. For the three other programming systems, we have measured the efficiency of the most relevant basic communication primitives.

Table 2: Null latency (in msecs) for Orca operation invocations.

	Original	Panda
ROI (RPC object invocation)	1.52	1.87
GOI (Group object invocation)	1.69	2.03

5.1 Orca

The performance measurements for Orca were carried out on the Amoeba processor pool, which contains 64 Micro-SPARC processors running at 50 MHz. Each processor is on a single board (a Tatung board, equivalent to the Sun Classic) and contains 32 MB of local memory. The boards are connected by a 10 Mbit/sec switched Ethernet.

The Panda implementation on Amoeba is described in [13]. Panda is implemented with communication protocols in user space. The protocols are implemented on top of Amoeba's unreliable unicast and multicast primitives. The RPC protocol is a 2-way stop-and-wait protocol (although Panda sometimes uses a larger packet size than that provided by the underlying hardware). The broadcast protocol is based on the sequencer protocols described in [10]. The Panda system uses a daemon thread that receives packets, reassembles them into messages, and then makes an upcall. The Panda system uses Amoeba's kernel threads; the Panda synchronization primitives (mutexes and condition variables) are implemented on top of Amoeba mutexes. We compare the performance of this system with that of the original (nonportable) Orca system, which was implemented directly on top of Amoeba [2]. The two systems use the same compiler, but entirely different runtime systems.

The basic communication primitive in Orca is object invocation. The ROI (RPC object invocation) benchmark measures the cost of remote object invocations on nonreplicated objects by executing an increment operation on a remote object containing a single integer. The GOI (Group object invocation) benchmark uses a replicated object shared by 16 processors, where two of the processors in turn perform an increment operation on the object.

Table 2 gives the null-latency for a single remote or group object invocation on the original and Panda-based Orca system. The two benchmarks have almost the same latency, due to the hardware support for multicast given by Ethernet. As can be seen, the Panda system has some overhead compared to the original system. The Panda-based system, however, is much more portable and also provides some more functionality. In particular, it does fragmentation for large messages, whereas the original system had an upper bound on the size of broadcast messages (and thus on the number of bytes that could be passed in a write operation on a replicated object). Also, the Panda system uses user-level communication protocols, whereas the original system uses kernel protocols. User-level protocols are in general more flexible. For example, the optimization with continuations described in Section 4.1 cannot be implemented with Amoeba kernel-level protocols, because Amoeba RPC requires the same thread that receives an RPC to also send the reply.

The overhead of the ROI benchmark is mainly due to an additional context switch at the client side. If the reply of the RPC comes back, the daemon thread has to do a context switch to the Orca thread that did the invocation. The overhead for GOI is mainly attributed to running the broadcast protocol (including the sequencer) in user space, so more crossings between user space and kernel space are needed compared to the original implementation [13]. In addition to these benchmarks, we have also run a large number of Orca applications on both systems. The results show that the Panda-based system achieves about the same absolute and relative performance.

Table 3: Null latency (in msecs) for PVM roundtrip messages.

	Original	Panda
Standard routing	6.891	-
Direct routing	2.535	3.645

5.2 PVM

We compare the performance of the original Solaris implementation of PVM with the performance of the Panda implementation. These measurements were performed on a collection of 80 MHz MicroSPARC-II workstation with 32 MByte of memory. All workstations run Solaris 2.4 and are connected by a 10 Mbit/sec Ethernet. The Solaris implementation of Panda uses Solaris threads and UDP/IP for interprocessor communication.

We used a simple ping-pong program to measure the roundtrip latency of PVM messages (see Table 3). For the original PVM system, we give two performance numbers. The latency for "standard routing" refers to the original Solaris implementation that routes all messages through daemon processes. The latency for "direct routing" refers to the same implementation, but in this case the benchmark programs call a PVM primitive that uses direct TCP connections between the application processes; messages are not routed through the daemons. Recall that the Panda implementation does not use separate daemon processes, so it always uses direct routing.

The original PVM system with standard routing is much slower than the Panda-based system. This is caused by the process switches and the copying needed to pass messages between application and daemon processes. With direct routing, however, the original PVM system is faster than the Panda-based system. This difference is mainly due to Panda's message delivery protocol, which is designed for multithreaded programs. Incoming messages are not received directly by an application thread, but are dispatched through an upcall thread. Therefore the ping-pong program on Panda incurs a context switch on both sides to pass control from the upcall thread to the main PVM thread. A context switch takes about 250 μs (i.e., 500 μs overhead in total). Furthermore, the usage of the thread-safe Solaris libraries adds another 300 μs to the roundtrip latency. The remaining overhead (310 μs) is the price we pay for resolving the differences between the PVM and Panda interfaces (e.g., FIFO versus LIFO packing).

5.3 SR

The performance measurements for the Panda implementation of SR were executed on two 70 MHz SPARCstation 5 workstations with standard 10Mbit/sec Ethernet adaptors, running Solaris. Each microbenchmark represents a different form of interaction between two virtual machines. The figures given for the original implementation come from MultiSR, the multiprocessor version of SR. MultiSR was used because it requires locking of internal data structures, which is also done in the Panda implementation.

The RPC benchmark measures the time it takes to invoke a null remote operation. On the remote virtual machine a new process is created to service the operation, which replies on completion. The rendezvous benchmark consists of a process invoking a remote operation that is serviced by waiting processes on the remote virtual machine. The asynchronous message passing benchmark measures the time required to send a null message from one virtual machine to another.

Table 4: Null latency (in msecs) for SR operations.

	Original	Panda
RPC, new process	4.53	6.68
rendezvous	4.71	7.74
asynchronous message passing	1.92	3.77

The performance differences between the original implementation and the Panda implementation are mostly due to thread creation and context switch times. The original system uses a custom user-level threads packages, whereas the Panda implementation uses Solaris threads. For example, process creation in the original implementation is 160 μs versus 688 μs in the Panda implementation. Also, a context switch in the original implementation is 63 μs and 119 μs in the Panda implementation. Additional overhead is attributed to the procedure call overhead to the Panda synchronization functions and the use of thread specific data to access internal information associated with each SR process.

6 Conclusions

We have described our experiences in using the Panda virtual machine for developing portable implementations of Orca, data parallel Orca, PVM, and SR. Each of the four resulting systems has been tested on at least two platforms (a cluster of Solaris workstations and the Amoeba processor pool); some of them were also tested on multicomputers like the PowerXplorer and CM-5. In our experience so far, adapting a programming system to the Panda interface takes at most a few months; porting the resulting system to other platforms on which Panda runs takes very little time.

We have analyzed the performance of the systems by comparing the Panda-based implementations with the original implementations. In general, the usage of Panda somewhat increases the latency of the basic communication operations. In many cases, the overhead is related to threads (thread creation, context switching, or thread-safety). Also, the Panda protocols run in user space, which has some overhead [13]. The great advantages of using Panda, however, are increased portability, modularity, and the flexibility of user-space protocols.

Acknowledgments

This research is supported in part by a PIONIER grant from the Netherlands Organization for Scientific Research (N.W.O.).

Many people contributed to the Panda project, including Frans Kaashoek (design of Panda), Ceriel Jacobs (Orca compiler), Rutger Hofman (Panda protocols). Jan Harkes (PVM implementation), Saniya Ben Hassen (data parallel extensions to Orca), Marco Oey (Amoeba and Solaris ports), Heinz-Peter Heinzle (Parix port), and Cees Verstoep (Myrinet port). We thank Ceriel Jacobs and Ron Olsson for their useful comments on the paper.

References

[1] G.R. Andrews and R.A. Olsson. *The SR Programming Language: Concurrency in Practice.* The Benjamin/Cummings Publishing Company, Redwood City, CA, 1993.

[2] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[3] R.A.F. Bhoedjang and K. Langendoen. Friendly and Efficient Message Handling. In *Proc of the 29th Hawaii International Conference on System Sciences*, pages 121–130, Maui, Hawaii, January 1996.

[4] R.A.F. Bhoedjang, T. Rühl, R. Hofman, K. Langendoen, H.E. Bal, and M.F. Kaashoek. Panda: A Portable Platform to Support Parallel Programming Languages. *Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 213–226, September 1993.

[5] The PORTS Consortium. PORTS: POrtable RunTime System. *documents available from http://www.cs.uoregon.edu/paracomp/ports/*, 1996.

[6] The MPI Forum. MPI: A Message Passing Interface. *Supercomputing'93*, pages 878–883, November 1993.

[7] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing (to appear)*, 1996.

[8] M. Haines, D. Cronk, and P. Mehrota. On the Design of Chant: a Talking Threads Package. In *Supercomputing '94*, pages 350–359, Washington D.C., November 1994.

[9] S. Ben Hassen and H.E. Bal. Integrating Task and Data Parallelism Using Shared Objects. In *10th ACM International Conference on Supercomputing*, pages 317–324, Philadelphia, PA, May 1996.

[10] M.F. Kaashoek. *Group Communication in Distributed Computer Systems.* PhD thesis, Vrije Universiteit, Amsterdam, December 1992.

[11] L.V. Kale, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. An Interoperable Framework for Parallel Programming. In *Proc. International Parallel Processing Symposium*, Honolulu, Hawaii, April 1996.

[12] F. Mueller. A Library Implementation of POSIX Threads under UNIX. In *Proc. USENIX Conference Winter'93*, pages 29–41, San Diego, CA, Jan. 1993.

[13] M. Oey, K. Langendoen, and H.E. Bal. Comparing Kernel-Space and User-Space Communication Protocols on Amoeba. In *Proc. of the 15th International Conference on Distributed Computing Systems*, pages 238–245, Vancouver, British Columbia, Canada, May 1995.

[14] W. Shu. Runtime Support for User-Level Ultra Lightweight Threads on Massively Parallel Distributed Memory Machines. In *Proc. Frontiers 1995*, pages 448–455, January 1995.

[15] V.S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[16] R. van Renesse, K.P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Comm. of the ACM*, 39(4):76–83, April 1996.