A Communication Kernel for Parallel Programming Support on a Massively Parallel Processor System

B.J. Overeinder J.J.J Vesseur F. v/d Linden P.M.A. Sloot
Parallel Scientific Computing & Simulation Group
University of Amsterdam, The Netherlands
e-mail: bjo@fwi.uva.nl
URL: http://www.fwi.uva.nl/fwi/research/vg4/pwrs/

Abstract

Portable parallel programming environments, such as PVM, MPI, and Express, offer a message passing interface that significantly differs in functionality provided by native parallel operating systems such as PARIX. The implementation of such a portable programming environment on top of the native operating system requires a considerable effort. To ease the porting effort, we have designed a Communication Kernel that offers a common base for the implementation of these environments.

The functionality of the Communication Kernel is close to the portable programming environments, and it is optimized to work efficiently with the underlying PARIX operating system. The current implementation of PVM profits from the efficient Communication Kernel as may future implementations of MPI and Express.

1 Introduction

The programming interface offered by portable parallel programming environments, such as PVM, MPI, or Express, creates a gap between the offered functionality in the parallel programming environment and the native parallel operating systems running on massively parallel processors (MPPs). This functionality includes for example various modes of point-to-point communication, collective communication, process groups, and process topologies. And although native operating systems support some of these features, higher level functionalities are often absent.

Because the required functionality is of generic interest to multiple programming environments, we have designed and implemented a generic programming interface, the Communication Kernel, that efficiently supports well-known portable parallel programming environments. The Communication Kernel provides a common basis for the design and implementation of the message passing interface, where the implementation intrinsics are clearly separated from the operating system dependent characteristics. This improves both maintainability and portability to, for example, newer versions.

In this paper we describe how the generic programming interface bridges the gap between PARIX functionalities and message passing interfaces. As a case study, we validated this Communication Kernel by implementing PVM on top of it. This PVM implementation is called PowerPVM to distinguish it from the public domain PVM version. In Section 2 the differences in functionality between PVM and PARIX are identified. Section 3 describes the Communication Kernel, and in Section 4 some performance results are presented.

2 Generic PVM versus Native PARIX

Different concepts and approaches to message passing has resulted in a multitude of message passing interfaces. In this section we identify the differences in functionality of the Parallel Virtual Machine system (PVM) [2] and the PARIX (PARallel extensions to UnIX) operating system [4]. These differences stem, among other things, from the fact that the PVM system was developed as a software framework for heterogeneous parallel computing in networked environments whereas PARIX was designed for tightly coupled parallel architectures.

2.1 Process Management

In PVM applications are viewed as comprising several sub-algorithms, each of which is potentially different in terms of it most appropriate programming model. Basically the Multiple Program – Multiple Data (MPMD) programming model applies to PVM. PARIX however, primarily supports the Single Program – Multiple Data (SPMD) programming model, although facilities are available for the MPMD paradigm.

The creation of a PVM task is accomplished with the `pvm_spawn` call. The spawned process starts an executable specified as an argument to the call. The node on which the task is started is specified by the user or is left to the PVM system for heuristic load balancing.

In the PARIX programming environment, program startup loads the same executable on all the nodes in the allocated partition of the MPP architecture. On the nodes itself, threads are the only active objects of an application. The only way of executing any processes in parallel on one processor is by means of threads.

2.2 Basic Message Passing

The PVM model assumes that any task can send a message to any other PVM task. The PVM inter-processor communication provides typed, asynchronous buffered communication primitives. These primitives are asynchronous blocking send, asynchronous blocking receive, and non-blocking receive varieties. In the PVM terminology, a blocking send returns as soon as the send buffer is free for reuse regardless of the state of the receiver. A non-blocking receive immediately returns with either the data or a flag indicating that the data has not arrived, while a blocking receive returns only when the data is in the receive buffer.

A great part of PVM's popularity is due to its support to receive unsolicited messages. That is, with a wildcard for sender and/or message type it is possible to receive *any* message, any message from a particular task, or any message of a particular type (does not care from which task). On the other hand, by specifying sender and message type one can receive a unique message. This strong typing enhances the correctness of the application.

The basic communication primitive in PARIX is synchronous non-buffered communication based on the virtual link concept. A virtual link provides a bidirectional, point-to-point connection between arbitrary threads within the parallel machine. On top of this basic communication layer, PARIX provides asynchronous non-buffered communication and mailbox based communication (close to the PVM communication model).

2.3 Group Communication

The programmability of massively parallel computations strongly depends on global communication and synchronization primitives such as broadcast and barrier synchronization, respectively.

PVM supports the concept of user named groups, where a task can dynamically join or leave a group at any time. Various collective communication primitives are defined on groups such as broadcast, barrier, or the reduce operation.

No similar group constructs are available in PARIX, although collective communication primitives are provided by a public domain library. These primitives work with index vectors containing the participating node numbers.

3 The Communication Kernel

The Communication Kernel (CK) levels the functionality gap between portable, parallel programming environments such as PVM and MPI, and PARIX. It provides generic communication and process management primitives found in parallel programming environments, but is still highly efficient implemented on the native PARIX operating system. In Fig. 1 a global overview of the layered design is shown.

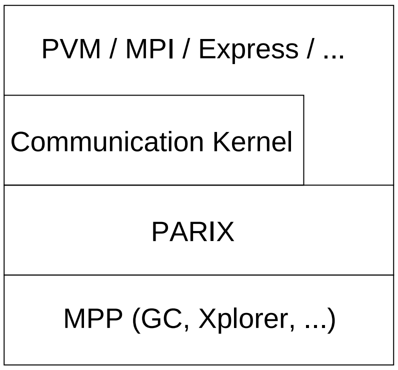


Figure 1: The Communication Kernel design overview.

With the design of the CK layer, the effort to efficiently implement the required functionality is invested only once. The contemplated portable message passing interfaces are implemented in terms of the CK primitives, resulting in a design with a clear abstraction from the operating system dependent characteristics of PARIX. Also, the advantage of this approach is that a performance enhancement in the CK layer directly translates in performance improvement of the portable programming environments.

In this section we will describe how the most conspicuous PVM characteristics are supported by the CK layer.

3.1 Process Management

The PVM run-time support is composed of a set of system processes called pvm daemons (pvmd). A pvmd is installed on each node in the parallel virtual machine.

In the CK, the run-time support resides at each node in the MPP architecture and is composed of one run server at the "master node" and spawn servers at each node. The run server coordinates the process creation requests in a similar way as the pvmd does. It allows user processes to create (spawn) processes at specific nodes in the parallel system, or on a suitable node determined by a round-robin scheduling algorithm augmented with load information.

The task of the spawn server at each node is the actual creation of a new process (thread). After a request for process creation from the run server, a new thread is created that starts the requested executable.

3.2 Message Passing

One of the key issues to level the gap between the two environments is the efficient support of inter-process communication facility characterized by typed, buffered asynchronous communication. The most obvious alternative would be the use of mailbox based communication supported by PARIX. One serious disadvantage however is the communication performance.

Therefore, we have designed and incorporated a new generic asynchronous buffered communication in the CK. It uses the synchronous non-buffered communication primitives of PARIX as its transport layer, and implements both blocking and non-blocking communication for user-level processes.

Messages can be accessed in three ways in the CK layer: any message, on sender, and on message type. The any message access corresponds with a PVM receive with a wildcard for the sender and message type, and will result in the first message in the receive queue, i.e., message delivery is *fair*. The message selection on sender identity is efficiently implemented by a hash table and a linked list for each sender in the receive queue. Upon a receive request for a particular sender, the first message in the linked list is retrieved and delivered. Message selection on message type cannot be implemented with hash tables easily, because the number of message types used is not known before hand, and therefore a proper hash function cannot be selected. Depending on the specification of the sender, the search for a message of a particular type starts at the beginning of the message queue (sender is wildcard), or at the beginning of the linked list of the specified sender.

3.3 Group Communication

In PVM a single task, the group server, is responsible for the dynamic group administration and for the coordination of barrier synchronizations. In the PVM design, the group server is a PVM task, using PVM communication primitives just like any other task. In our implementation, the group server is also a task but as part of the run-time support system. This design allows a more efficient implementation of the group server that directly interacts with the Communication Kernel.

For collective communication we have implemented an optimized multicast with hypercube routing over the processor grid.

Consider an N-node partition in the parallel machine, and that the addresses of the nodes can be coded in n-bits. In the first step, node 0 communicates with the node which differs in the most significant bit B_n. In the second step, both node 0 and node 2^{n-1} possesses the information and communicate this with the nodes which differ in the second most significant bit B_{n-1}, etc., etc. In this ways, the multicast messages are propagated to all the nodes in $\log_2(N)$ steps, see Fig. 2.

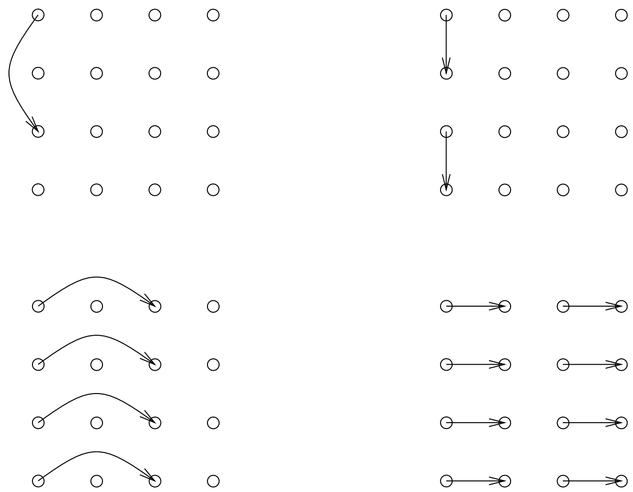


Figure 2: Hypercube routing by most significant bit over a mesh.

Another advantage of this approach is that with each step, the process grid is segmented in partitions that localizes the communication within the partition. This eliminates the problem of contention in the last steps of the multicast where all the nodes are actually communicating with each other.

4 Results

The well-known "ping-pong" experiment is performed to measure the communication performance of the CK layer and the PowerPVM implementation on top of it. In Fig. 3 the performance of the three message passing layer are depicted in a log-log plot. One can see that the PowerPVM performance is close to the performance of the CK layer. However, the difference in performance between the CK layer and PARIX is prominent: the loss is due to the functionality offered by CK.

The performance loss between PowerPVM and CK is mainly a result of the multiple send/receive buffer management in PVM. Every message sent and received must be copied to and fro. In order to compare the PowerPVM send and receive primitives with the underlying CK send and receive primitives, we have implemented a second PVM version of the ping-pong experiment similar to the CK version. By calling `pvm_setsbuf` the receive buffer becomes the new send buffer which is sent back immediately. Thus, the measurements do not include the unpacking and packing of data.

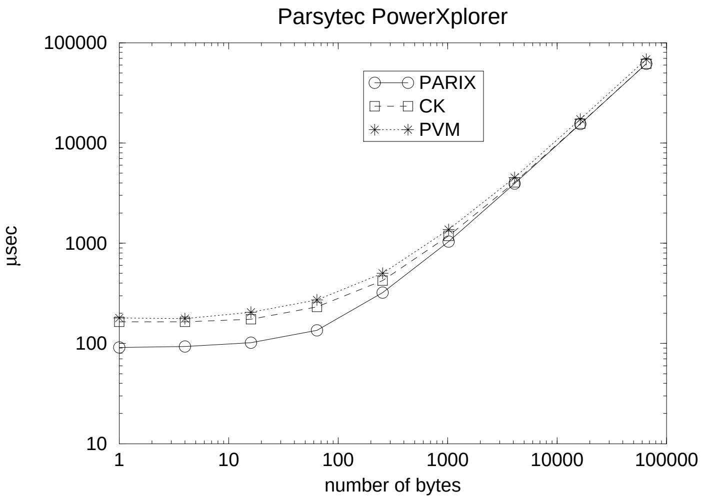
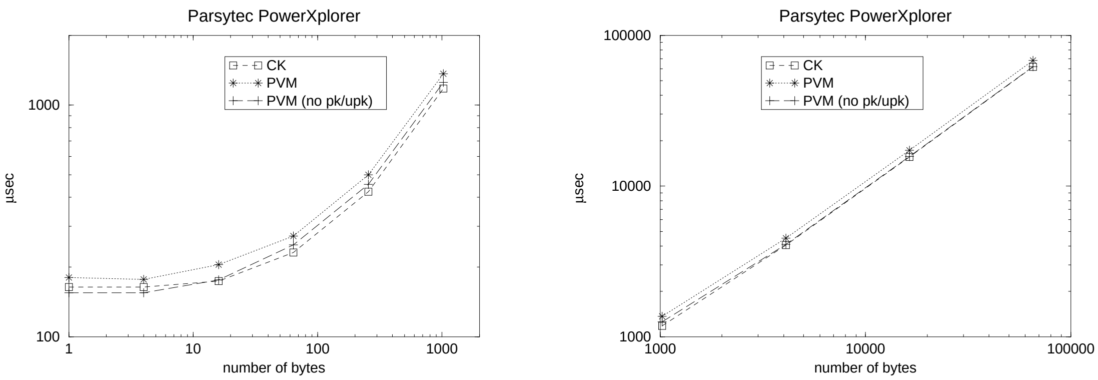


Figure 3: Communication performance of PARIX, CK, and PowerPVM on Parsytec PowerXplorer.

For clarity, Fig. 4(a) shows the performance of short messages ($\leq$ 1024 bytes), and Fig. 4(b) of long messages. For very short messages, the performance of PowerPVM without pack/unpack is even better than the CK layer. This is caused by a convenient delay introduced by PowerPVM such that the message arrives *just in time* at the receiver, saving the buffering in the CK layer.



(a) Small messages.

(b) Large messages.

Figure 4: Performance comparison of CK and PowerPVM.

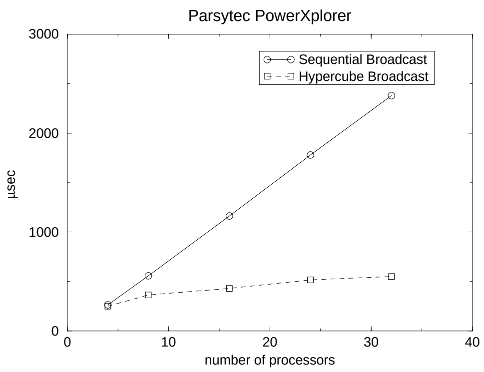
The overhead for PowerPVM measured on the Parsytec PowerXplorer is shown in Table 1, which shows latencies for PARIX 1.2, the CK layer, and the two PowerPVM versions, as well as the achieved throughput of the systems.

The broadcast benchmark (see Fig. 5(a)) shows the gain of *hypercube* routing over *star* routing (i.e., one task sending all the messages) in global communication. From the figure one can see that with star routing the broadcast completion time grows linearly with the number of tasks, while the hypercube routing obeys the $\log_2(N)$ time complexity.

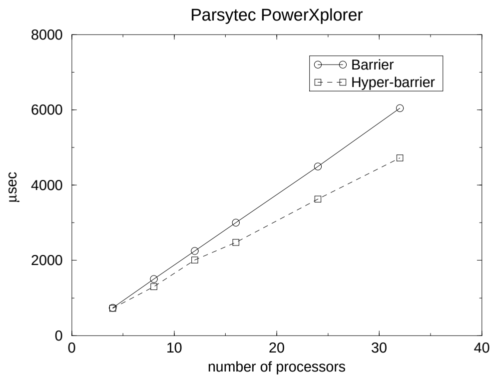
	Latency μsec	Throughput (Kb/sec)
PARIX	91	1038
CK	164	1036
PowerPVM	167	940
PowerPVM*	155	1033

Table 1: Communication latency and throughput measured on the PowerXplorer. * is version with no pack/unpack.

Within the barrier benchmark (Fig. 5(b)) the performance gain is less explicit. The advantage of concurrency in the hypercube routing scheme is reduced by the forced synchronization within the barrier primitive, but it performs still significant better than the star routing alternative.



(a) Broadcast communication.



(b) Barrier synchronization.

Figure 5: Performance of global communication primitives: sequential versus hypercube routing.

A more elaborate performance comparison between PVM and PARIX from an application perspective can be found in the paper by Hoekstra et al. [3].

5 Conclusion

We have developed a generic Communication Kernel that supports popular portable parallel programming environments such as PVM, MPI, and Express, where our aim was to integrate the best of both worlds: the full functionality of portable programming environments and the speed of native environments such as PARIX. The results clearly indicate the feasibility of our approach.

In the near future we will complete a full implementation of MPI [5] and Express [1] on top of the CK layer. The design of these environments will be in the same way as with the PowerPVM implementation. Enhancement and extension to the CK layer will continue.

References

[1] J. Flower and A. Kolawa, "Express is not just a message passing system: Current and future directions in Express," *Parallel Computing*, vol. 20, no. 4, pp. 597–614, Apr. 1994.

[2] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, "PVM 3 user's guide and reference manual," Tech. Rep. ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee, May 1994.

[3] A. G. Hoekstra, P. M. A. Sloot, F. van der Linden, M. van Muiswinkel, J. J. J. Vesseur, and L. O. Hertzberger, "Native and generic parallel programming environments on a Transputer and PowerPC platform," Accepted for publication in *Concurrency: Practice and Experience.*

[4] *PARIX 1.2 Documentation*. Parsytec GmbH.

[5] D. W. Walker, "The design of a standard message passing interface for distributed memory concurrent computers," *Parallel Computing*, vol. 20, no. 4, pp. 657–673, Apr. 1994.