Message-passing over shared memory for the DECK programming environment*

Rafael B. Ávila† Caciano Machado‡ Philippe O. A. Navaux§
Parallel and Distributed Processing Group
Instituto de Informática — UFRGS
Caixa Postal 15064
91501-970 Porto Alegre — Brazil
Phone: +55 51 3316-6165 Fax: +55 51 3316-7308
E-mail: {avila,caciano,navaux}@inf.ufrgs.br

Abstract

Message-passing is a representative communication model in today’s parallel and distributed programming, and should be efficiently supported even for multithreaded-only parallel programs. This paper describes the design and implementation of a communication mechanism which emulates message passing on top of shared memory for multithreaded applications. The mechanism is implemented in the DECK parallel programming environment and a performance analysis is presented.

Keywords: cluster computing, multithreading, parallel programming environment, message passing, shared memory.

1. Introduction and context

Message passing has been established, in the last years, as a de facto standard in parallel and distributed programming, mainly due to the popularization of workstation networks, or clusters, as parallel machines, programmed via libraries such as PVM [5] or MPI [11]. Though more “difficult” to use than shared memory, people have become accustomed to employing it. On the other hand, considering the availability of the not less popular SMP machines, featuring 2, 4 or even 8 processors in a single PC, multithreaded programming [10] is also significant and has become a key point to achieve top performance when combining computation and communication.

Our work is motivated by the fact that, even if shared memory is available for communication between threads, message passing might still be attractive, since programmers are used to it. Also, existing applications with distributed nature could be easily ported to a multithreaded environment.

But if the message passing subsystem is not aware of the possibility of multithreading, the resulting performance may be a disaster. Since communication must take place locally, it is desirable that shared memory be used for communication, instead of going the normal way through the operating system down to the network card and bouncing back to the application.

For these reasons, we present in this paper a mechanism for message passing emulated in the local memory of a node, for communication between local threads. Moreover, to the difference of existing implementations of such functionality, our implementation is based on chained lists of messages, which presents interesting results. This mechanism has been incorporated in the DECK parallel programming library, and a performance analysis has been made.

The paper is organized as follows: Section 2 presents an overview of DECK and its main features; Section 3 describes the implementation of MP on top of SHMEM and Section 4 presents an evaluation of it; finally, Section 5 poses some comparisons to similar implementations and Section 6 concludes the paper.

2. The DECK environment

DECK—Distributed Execution and Communication Kernel [2]— is a programming environment aimed at cluster computing. It features the most common resources for parallel programming and supports a variety of communication technologies such as Myrinet [3] and SCI [9] and respective protocols.

---

* This work was partially supported by project FINEP/CTPetro no. 65.00.0363.00 and by grants from CNPq and CAPES.
† PhD student
‡ Undergraduate student, research assistant
§ Professor, Dr. (Institut National Polytechnique de Grenoble, France, 1979)
The API of DECK provides 5 abstractions that we consider basic for any parallel application: threads, mutexes, semaphores, mailboxes and messages. Additionally, DECK still provides some features which depend exclusively on the basic abstractions, and thus are platform-independent. Currently, such features include collective communication, multithreaded servicing (pool of threads) and condition variables. These features are internally organized in a two-layer structure as shown in Figure 1. The basic abstractions form the lower layer, which we call µDECK. The other features form the upper layer and are selected at compile time, possibly being empty.

Threads and synchronization are based on the standard POSIX Threads implementation [10], and make direct use of its functions. Traditional primitives such as thread_create() and mutex_lock() are provided. The idea in DECK, however, is to minimize the complexity of POSIX Threads’ functions. For example, threads in DECK only exist in the attached state.

Inter-node communication in DECK is based on a mailbox abstraction. In order to communicate with another node, a thread must own a reference to a remote mailbox and post a message in it. Such message is later retrieved on the remote node by the thread owning the mailbox. Message posting has always asynchronous (non-blocking) semantics (except for flow-control constraints), and message retrieving is always synchronous (i.e. it blocks the calling thread until the message arrives.)

Communication is the most active development part of DECK. Since its first implementation for UDP sockets, in 1999, DECK has been ported to TCP, BIP [13] for Myrinet, and SISCI [7] for SCI. All of these implementations are freely available for download from the project’s homepage at gppd.inf.ufrgs.br/projects/mcluster. Results on the performance of DECK for these systems can be found in [1] and [4].

3. Message passing over shared memory in DECK

As stated before, the goal of our work is to use memory as a channel for the exchange of messages between threads. This situation is illustrated on Figure 2. In order to implement this mechanism on DECK, we have designed new post() and retrieve() functions that manipulate messages exclusively through shared memory. These new routines are coupled in the original post and retrieve primitives, which have been adapted to recognize when a thread sends/receives a message to/from another thread in the same node and, in this case, activate the local-delivery semantics.

To the difference of existing systems, the proposed shared-memory message passing mechanism was implemented using FIFO queues. The implementation uses a mechanism that dynamically allocates memory for messages in the post operations, and another mechanism to take advantage of the already allocated message structures and buffers to minimize memory allocation operations. This is useful because of the potentially heavy dynamic memory allocation process. As a consequence, it is best suited for applications that present a “pattern” in the posting and retrieving of messages and their sizes. If the application does not vary the size of messages too often, the reallocation operations will be minimized and performance will be improved.

The implemented mechanism dynamically manages two message queues, for posted and retrieved messages respectively, for each mail box referenced at least once. This means that when the application is started the mail box list of the node is empty, and only when a message is firstly posted to a mail box a descriptor for it will be created.

The mail box structure is a descriptor which contains information about the message FIFOs, along with the name of the mail box and a link to the next mail box in the list. The two queues are used for posted and retrieved messages. When a message is posted, it is added to the posted queue, being later removed when the corresponding retrieve occurs (Figure 3). The retrieved queue is used to take advantage of the already allocated message buffers. When a message is retrieved its message structure and buffer is moved to the end of the retrieved queue in order to be used by subsequent post operations.

All the mail boxes and message structures are accessed by a global reference to the mail box list. Concurrency for
the structures access is controlled by DECK mutexes (with spinning implementation).

The representation of messages in memory consists of a dynamic allocated structure which contains control information and a buffer which contains the data to be exchanged (payload). Furthermore, the message structure contains a link to the next message on the FIFO or a NULL value if it is the last in the queue.

Here is a simple description of the routines work:

- **Post**

1. First the `deck_mbox_post()` function tests if the destination of the message is in a thread on the local node and, if true, the shared memory routines are called
2. The local post function tests if the target mailbox exists and if not it waits for a post operation that will create the mailbox and post the message
3. If the mailbox has message structures in the message queue, the function recovers the data associated with the first message of the queue if it exists
4. The data and control information are copied to the user message
5. The retrieved message object is linked in the retrieved message queue to be used by other post operations.

- **Retrieve**

1. As in the case of the post function, `deck_mbox_retrv()` tests if the sender of the message is in the local node
2. The function tests if the mail box exists and if not it waits for a post operation that will create the mail box and post the message
3. If the mailbox has message structures in the message queue, the function recovers the data associated with the first message of the queue if it exists
4. The data and control information are copied to the user message
5. The retrieved message object is linked in the retrieved message queue to be used by other post operations.

### 4. Evaluation

In order to evaluate the performance of the DECK shared-memory message passing functions, we have compared the exchange of messages of different sizes in three situations:

- between two DECK threads using the implementation just described
- between two DECK threads using our TCP implementation
- between two MPI processes on the same node

In the case of MPI we had to make use of two processes since the library does not natively support threads. On all experiments we have used a Dual Pentium III 500MHz with 256M of RAM. The operating system is GNU/Linux with kernel 2.4.12 and glibc 2.2.3, and the MPI implementation is MPICH 1.2.2. Additionally, every result presented has been taken as a mean of at least 1000 repetitions.

We have measured the raw performance of the three implementations for the exchange of messages varying from 1 byte up to 8 Mbytes. Figure 4 presents the obtained performance in terms of latency (measured as half of the round-trip time). These results show a significant overall improvement in our implementation in relation to TCP or MPI. DECK has been able to reach only 5 \( \mu \)s latency, against 50 \( \mu \)s of TCP and 70 \( \mu \)s of MPI. We understand that the gain in performance comes from the use of threads, not possible in MPI, and the reuse of buffers in the message queues, to our knowledge not used in neither MPI or the TCP implementation of the Linux kernel. It is also possible to notice that local TCP message exchanges in the kernel are greatly optimized in the 2.4 series. We have made experiments with 2.2 kernels and obtained results that are at least two orders of magnitude higher. In that case, the performance obtained by the local post and retrieve functions is even more significant.
Figure 4. Results for message exchange latency.

The benefits from the shared-memory message passing functions are more evident when we analyze the achievable bandwidth, shown in Figure 5. The gain in performance is specially evident for message sizes between 32 and 64 Kbytes. The peak bandwidth is achieved for messages of 32 Kbytes, delivering about 125 Mbytes/s. The peak values for TCP and MPI are respectively 112 and 52 Mbytes/s. We believe the same reasons for bad performance apply here, specially in the case of MPI.

5. Related work

Two available implementations of similar mechanisms for memory-based communication have influenced our work more directly, which we present in this Section.

The first one is the `chshm` implementation of MPICH[12]. This is a device layer for shared memory message exchange between MPI processes running on the same node. The implementation uses three different protocols for message passing, depending on the message size. The first protocol, dedicated to small messages, uses a pre-allocated area in memory to the send and receive operations. The second protocol is responsible for medium-size messages and uses a dynamic allocation mechanism for data manipulation. The last protocol, used for large messages, imposes a rendez-vous semantics for the send and receive operations, copying the message directly to the destination buffer.

Our implementation is similar to the second protocol of MPICH, but the difference lies on the fact that message buffers are reused in DECK, and in MPI they are freed after the corresponding receive is performed. This behaviour imposes a higher overhead, but may be more balanced for applications with irregular communication pattern.

The second implementation is BIP-SMP [6], of the University of Lyon, France. BIP-SMP is a protocol for intra-node communication in clusters of SMPs. The interesting characteristic of this implementation is the protocol used for the exchange of large messages. The exchange of small messages is similar to that used in DECK and the MPICH `chshm`. The data exchange of large messages is done by direct memory copy, i.e., the message is copied from the user space of the sender process to the user space of the receiver process. This is possible because the researchers have implemented a kernel module for Linux that allows data exchange between the processes of a node.

In their article the authors show many performance results for BIP, MPI and other message passing libraries like GM [8] employing the mentioned protocol and specific implementations. A mostly interesting result, in relation to the work presented here, is mentioned for the `chshm` device, designed for MPICH, where latency and bandwidth reach values of 2.4 µs and 100 MBytes/s respectively. Pure BIP-SMP is capable of reaching 1.8 µs and 160 MBytes/s between two processes on a SMP node.

6. Conclusions and future directions

The performance obtained by the message passing emulation mechanism presented in this paper has been considered very good, being able to present results which are comparable and even outperform those presented by traditional programming interfaces such as TCP and MPI. For example, in the case of latency, the results obtained for DECK represent 10% of TCP and only 7% of that of a traditional MPI implementation. Similarly, we have shown that other research projects have been able to achieve even better results when using dedicated features. As a general conclusion, and for the reasons exposed in the introduction, posing the importance of message passing primitives in parallel and distributed programming, we have considered this work a significant contribution to the DECK environment.

Our next steps will be targeted at providing a shared
memory based communication mechanism for DECK processes running on the same SMP node, similarly to that of MPICH’s ch_shmem. Also, we plan to run some experiments comparing the performance of DECK with that of local communication when using specific communication libraries such as BIP and SISCI.

7. Acknowledgements

The authors would like to thank Maurício Pilla for his valuable observations on process and thread scheduling.

References


