DECK: an environment for parallel programming on clusters of multiprocessors

Marcos Barreto[1*], Rafael Ávila[1†], Fábio Oliveira[1‡], Ricardo Cassali[1§], Philippe Navaux[1¶]

[1] Federal University of Rio Grande do Sul
Institute of Informatics
Group of Parallel and Distributed Processing
Av. Bento Gonçalves, 9500 Bloco IV
PO Box 15064 — 90501-910 Porto Alegre, Brazil
E-mail: {barreto,avila,fabreu,cassali,navaux}@inf.ufrgs.br

Abstract— **As cluster-based architectures are becoming a well-established platform to the development and execution of parallel applications which demand for high performance, new requirements are imposed to the software used in such architectures. Within this context, we have developed DECK, an environment conceived to allow the development of parallel applications on top of this new architecture. This work presents the internal structure and the functionalities of DECK, as well as a performance evaluation of its two available implementations.**

Keywords— **Cluster computing, Parallel programming environments, DECK, Performance evaluation**

I. INTRODUCTION

Due to the appearance and availability of fast communication networks, such as Fast [IEE95] and Gigabit Ethernet [CUN99], Myrinet [BOD95] and SCI [IEE92], the scenario of parallel and distributed programming on clusters of commodity workstations has changed. Communication libraries such as PVM [GEI94] and MPI [MPI94], widely used for this purpose, have became inefficient due to their weak capacities to explore the underlying high performance networks.

New proposals in terms of communication protocols have arised with the insigth of providing high performance for applications that run over cluster-based architectures. Some of these proposals are BIP [PRY98], Fast Messages [PAK96] and GM [GM99] libraries, as well as the VIA [VIA99] specification. Since the objective of these proposals is to serve as a basic communication layer, several parallel programming environments have been developed on top of them, aiming to provide high performance and, at the same time, to hide the functional details of the underlying architecture. In a similar manner, existing communication libraries, such as PVM and MPI, have been adapted to use these new libraries.

According to the aforementioned trend, this work presents a parallel programming environment, called DECK (*Distributed Execution and Communication Kernel*), which was developed for cluster-based architectures. Like other distributed communication kernels, such as PM^2 [NAM96], Athapascan0 [GIN97, CAR98] and Nexus [FOS94, FOS96], DECK integrates communication and multithreading and also achieves high performance over cluster-based architectures. However, DECK has some characteristics that differs it from the others environments: the provided abstractions are more simple, it foresees the support for multiple clusters and its layered structure allows the use of different communication technologies.

The results obtained with DECK have shown true gains in terms of latency and bandwidth when compared with other distributed communication kernels, for both available implementations (Fast Ethernet and Myrinet). On the other hand, the implementation on top of Myrinet has presented some problems regarding large messages transmission, leading us to a deep study on the BIP [PRY98] library and to the development of a flow control protocol.

This paper is organized as follows: section II presents a description of the DECK environment, its internal structure and functional model; section III describes the two available implementations of DECK, for Fast Ethernet- and Myrinet-based clusters; in section IV, a performance evaluation of both versions is exposed; section V brings some comments on similar environments and, finally, in section VI, some considerations about the development of this work are pointed out, as well as the current research work.

II. DECK

The DECK environment is composed of a runtime system and an user API which offers communication and multiprogramming resources to allow the development of parallel and distributed applications over cluster-based architectures. The original model of DECK was firstly described in [BAR98] and, from this model, two implementations were made in or-

*M.Sc (2000); Ph.D student at PPGC/UFRGS.
†M.Sc (1999); Ph.D student at PPGC/UFRGS.
‡B.Sc (1998); Master student at PPGC/UFRGS.
§Assistant Researcher – PIBIC/UFRGS.
¶Ph.D (INPG, Grenoble — France, 1979); Professor at PPGC/UFRGS.

der to address two aspects:

- provide support for DPC++, which is a distributed object-oriented language that allows intra-object concurrency [ÁVI99];
- serve as programming environment for the MultiCluster project [BAR00], whose main goal is to join different cluster architectures, such as Myrinet and SCI clusters, and provide an efficient and simple programming interface for this integrated platform.

A. Programming model

DECK follows the SPMD programming model, in which there is one process copy on each node used by the application. Each process can create many computational threads, which communicate with each other by means of the shared memory space sustained by the process. Data exchange between threads running on different processes must be realized through explicit message passing primitives.

B. Software structure

DECK is divided in two layers, one called **μDECK**, which directly interacts with the underlying operating system; and a **services layer**, where more elaborated resources are made available. Figure 1 shows the layered structure of DECK.

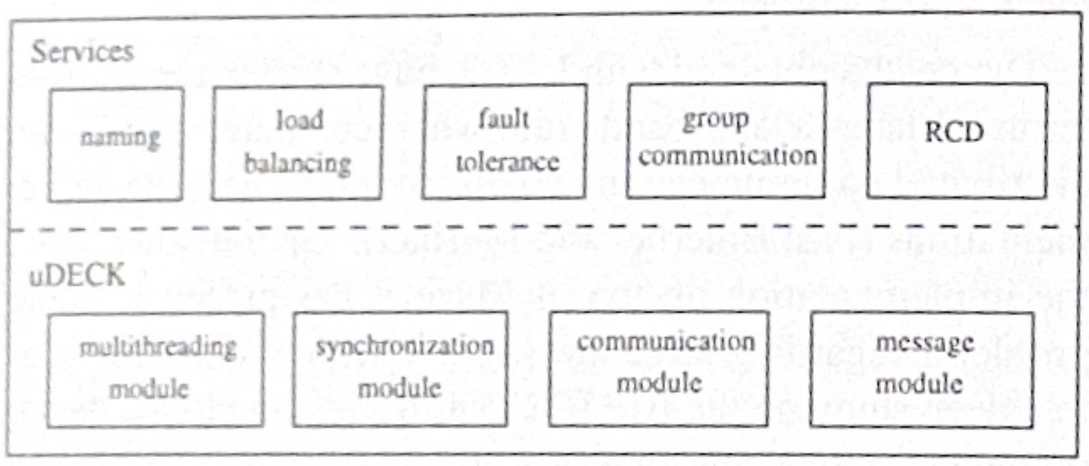


Fig. 1. Internal structure of DECK.

μDECK is the platform-dependent part of DECK. This layer implements the four basic abstractions provided within the environment: *threads, semaphores, mailboxes* and *messages*. Each of these abstractions is treated by the application as an object, and has associated primitives for proper manipulation.

Threads are used to allow the application to specify different execution flows within a same process, whereas semaphores are used to coordinate the concurrent accesses to shared objects. It is important to notice that threads and semaphores are both local resources, i.e., DECK does not allow remote thread creation and remote access to semaphores.

Messages can be posted to or retrieved from mailboxes, which are abstractions used to establish point-to-point communication. Only the creator of a mailbox is able to retrieve messages from it, but any other thread knowing the mailbox can post to it. To use a mailbox, the creator must register it in a name server. There are two ways to obtain a mailbox address: fetching it in the name server or receiving it in a message.

The DECK's upper layer is responsible for furnishing more sophisticated services, which are divided into two categories: **global services** and **local services**. Global services correspond to services always used by any application, for example, the name server; **local services** correspond to services which are used only if needed by a specific application, for instance, fault tolerance and group communication. Once available, the user will be able to choose when he wants to use these services, by means of compilation options.

The name server's functions are to store the location of mailboxes and to allow the user to assign a logic name to each mailbox used by his/her application.

Concerning local services, the DECK model foresees the availability of a load balancing mechanism in order to better distribute the workload across the cluster nodes. Fault tolerance is also defined with the aim of allowing the safe execution of complex applications, such as parallel simulations, which usually have high execution times. Additionally, a group communication protocol is defined in order to offer a set of basic collective communication primitives that provide the user with the necessary toolbox to develop more elaborated message difusion and/or atomic agreement protocols.

Within the scope of the MultiCluster project, DECK defines a service called RCD (*Remote Communication Daemon*) which is in charge of routing messages between threads and/or processes running on different clusters and solving problems related to different data representation, in the case of heterogeneous clusters. As the goal of this project is to join clusters which are different in terms of communication networks (Myrinet + SCI, for example), the RCD will allow us to develop independent implementations of DECK and, after, join them in a simple way through this inter-cluster communication mechanism.

III. Implementation

The implementation of DECK was carried out in two phases, as described in figure 2. The first version of DECK corresponds to the implementation of the μDECK layer and the name server for Fast Ethernet clusters. This implementation was focused on validating the modular structure of the environment and on the definition of a basic programming interface to allow the development of some services, such as load balancing and group communication.

Both services are ongoing tasks: load balancing will be used by the DPC++ language in order to choose the best location (node) every time a distributed object must be created; the group communication protocol is in a stage of evaluation, being used to solve linear equation systems [CAS00].

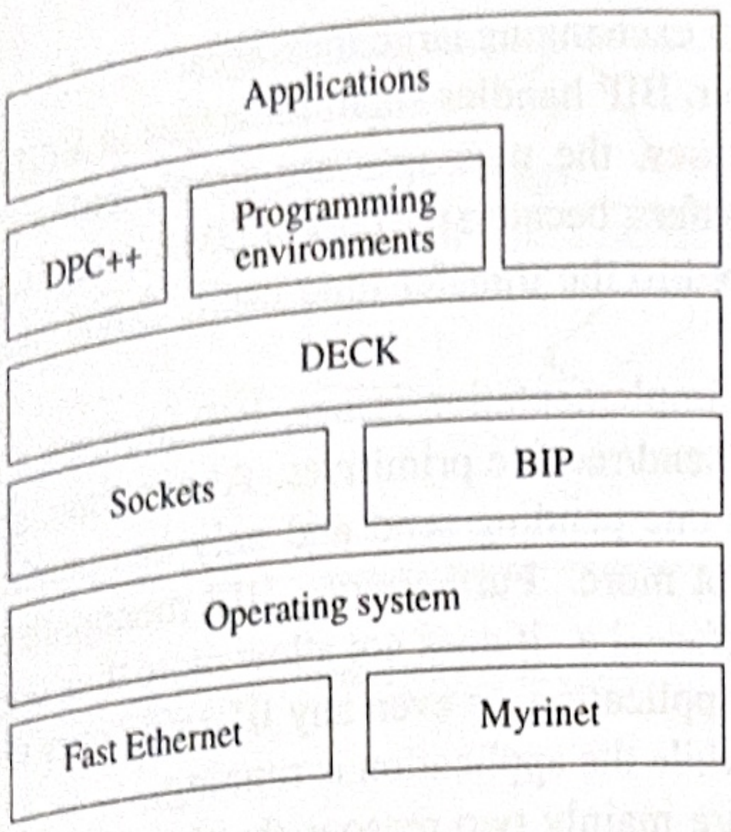


Fig. 2. Implementation of the DECK environment.

After this, a new version of DECK was developed with the objective of efficiently explore the Myrinet network in order to provide high performance. To accomplish such goal, the BIP protocol [PRY98] was chosen as communication layer and the DECK/Fast Ethernet version was adapted to use it. This section presents some considerations about both implementations.

A. The Fast Ethernet implementation

A.1 The μDECK layer

The primitives related to multiprogramming and synchronization are trivial, based on the Pthreads package [IEE95a], which provides threads, *mutexes* and conditional variables used to implement the DECK's thread and semaphore abstractions. Although trivial, some considerations about these primitives are necessary: threads are always created in a *detached* state, to better use the system resources, and their scheduling is made at the system level. Regarding synchronization, it is important to say that semaphores are always local to a specific process and can not be accessed by remote threads.

The mailbox implementation is based on Berkeley sockets, with a datagram semantics. The creation of a mailbox allocates a new socket descriptor, in association with the communication port and the machine address on which this operation was performed. Mailboxes can be sent to remote nodes by means of messages. A mailbox received through a message only accepts `send` operations. In this case, the mailbox descriptor is not valid because it corresponds to a remote resource; hence, a new socket is created and the previously stored address is used to find the destination node.

Figure 3 illustrates the primitives related to mailboxes. The primitives `deck_mbox_post()` and `deck_mbox_retrv()` corresponds to the traditional `send` and `receive` primitives, respectively. For both DECK implementations, the send is asynchronous and the receive is synchronous. The primitives `deck_mbox_bind()` and `deck_mbox_fetch()` are used, respectively, to register mailboxes in the name server and to get mailboxes descriptors from it.

```
deck_mbox_create();
deck_mbox_destroy();
deck_mbox_post();
deck_mbox_retrv();
deck_mbox_bind();
deck_mbox_fetch();
```

Fig. 3. Mailbox interface.

```
deck_msg_create();
deck_msg_destroy();
deck_msg_pack();
deck_msg_unpack();
deck_msg_clear();
deck_msg_reset();
```

Fig. 4. Message interface.

Once it has been created, a mailbox must be registered in the name server so that other processes can find it to establish communication. During the execution of a `deck_mbox_fetch()` primitive, the environment guarantees that a desired mailbox will be found, by executing a `deck_thread_sleep()` primitive until the completion of the mailbox registration by its owner.

The primitives related to message handling are shown in Figure 4. The primitive `deck_msg_create()` is used to create a message, allowing the user to specify an existing buffer (of his own); if the user does not do so, the environment automatically creates a data buffer for his message. DECK allows the user to pack and unpack messages, using in such operations, besides the common types, the mailbox type — `deck_mbox_t` — and even the message type — `deck_msg_t`. The primitive `deck_msg_clear()` is used to clear the message buffer, whereas the primitive `deck_msg_reset()` is used to arrange the cursor within the buffer, during packing and unpacking operations.

A.2 The services layer

The name server is implemented as a dedicated thread which runs always in the first node of the cluster and has a well-known mailbox within the system. It executes a loop until the end of the application, processing requests for registering a mailbox or fetching for a given mailbox. It is important to notice that the primitives for mailbox registration and search, respectively `deck_mbox_bind()` and `deck_mbox_fetch()`, are implemented by the communi-

cation module and constitute the unique access interface to the name server from the user's point of view.

B. The Myrinet implementation

DECK/Myrinet was implemented on top of BIP (*Basic Interface for Parallelism*) [PRY98] messaging system, which is a very efficient protocol specific for Myrinet-based clusters. Basically, BIP supplies a set of message passing primitives with blocking and non-blocking semantics. The exchange of messages in BIP counts on the idea of *tags* (queues for receiving messages). After a call to the initialization function (`bip_init`), all the tags in the calling process are configured with a default number of buffers for small messages. However, the programmer can modify this configuration independently for any tag, calling `bip_tag_init`, by means of which it is possible to define that a specific tag must have a given number of buffers with a specified size. Each node has `NTAGS` tags, whose numbers range from 0 to `NTAGS` − 1.

The BIP API's main focus is on a set of blocking and non-blocking send/receive primitives. The send primitives usually need the target node, the target tag and a buffer message as arguments, whereas in the receive side one must provide the receive tag (the tag that will store the incoming message), a message buffer (where the message will be stored after receive completion) and the size of the message. Some receive functions can identify the sender node, or even if the incoming message is small or large (this distinction is very important).

At the BIP level, the behaviour of the communication primitives distinguishes between small and large messages. Large messages transmissions have a *rendez-vous* semantics, in which a receive must be posted before the respective send. On the other hand, small messages are stored in tag's buffers on the receiver node, so that a send can be posted before the target node is ready to receive a message. The constant `BIPSMALLSIZE` defines the limit between small and large messages.

The aforementioned difference is due to the way BIP deliveries messages. When dealing with large messages, BIP decomposes the message into packets of equal sizes (which depend on the total length) and executes a pipelined transmission between the communicating nodes, according to the following concurrent steps: (1) DMA transfer from main memory to the network interface's SRAM memory; (2) transfer of the previous packet from network interface's SRAM memory to the network. Of course, at the receiver node there are two concurrent steps symmetrical to these ones. Hence, if the message is split into more than 4 packets, there will be a full pipeline transmission, improving the overall performance. The drawback of this strategy, however, is that BIP programmers must supply a flow-control protocol of their own, when exchanging large messages.

However, BIP handles small messages in a different way. In such cases, the messages are simply buffered, without DMA transfers, because the cost of DMA setup is prohibitive if compared to the transfer time itself. Small messages are not split.

Due to implementation issues, BIP imposes a constraint regarding send/receive primitives. At one time, a node may have only one pending send and only one pending receive per tag, not more. Furthermore, BIP monopolizes the network interface, i.e., it does not allow more than one independent BIP application, or even any IP traffic over the Myrinet network while the application is running.

There are mainly two reasons that have led us to choose BIP among various Myrinet messaging systems currently available. First of all, BIP follows a very clear and straightforward message-passing API, without introducing unusual paradigms or concepts. Second, besides the simple API, BIP can delivery messages with very low latency (about 5 μs) and high bandwidth (about 120 MB/s). These outcomes are achievable thanks to BIP's very specialized design that absolutely kept track of Myrinet hardware capabilities.

The process of porting the first version of DECK to use BIP has demanded changes on two of the provided abstractions: *messages* and *mailboxes*. Additionally, as the BIP protocol does not implement any kind of flow-control mechanism, this new implementation of DECK has to deal with problems related to large messages transmission, as it will be described in the next sections.

B.1 Message creation

As mentioned in section A.1, the user can pass a buffer of his own when he creates a message; otherwise DECK provides one by dynamic allocation. In the BIP implementation, a restriction has to be imposed relating to buffer alignment in memory: BIP requires that user buffers be aligned in a 4-byte base, which is a *word* in the BIP terminology. Thus, DECK has to check, in the case of user-supplied buffers, if the proper alignment has been respected. When the user does not provide a properly aligned buffer, the message creation fails.

B.2 BIP tags and DECK mailboxes

As message exchange in BIP is based on reception queues labeled with tags, our mailbox implementation assigns a specific tag to each created mailbox, as well as the identification of its creator node. In contrast to the Fast Ethernet mailbox implementation, in which for each mailbox received in a message a new socket must be created, there is no need for such an expensive operation in the Myrinet implementation. The tag numbers within all received mailboxes are always valid and do not require an additional creation operation.

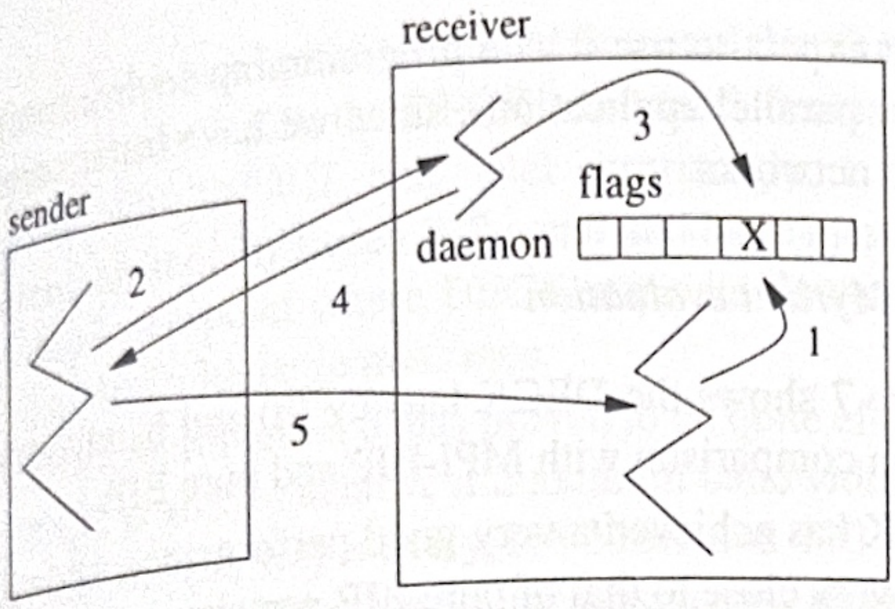


Fig. 5. *Rendez-vous* protocol used by DECK/Myrinet.

B.3 Dealing with large messages

The main concern in using BIP to implement μDECK is related to the exchange of large messages, since BIP requires an user-level flow-control mechanism in this case.

To guarantee the *rendez-vous* semantics imposed by BIP, the implementation of DECK on top of it uses request messages to assure that a large message will be transmitted by the sender only after the receiver has declared a reception buffer for it. The flow-control protocol is implemented with the help of a dedicated thread which runs in each node and has the responsibility of verifying the availability of reception buffers for large messages, as illustrated in Figure 5.

When a thread posts a small message, this one is sent directly to the destination mailbox, since BIP is able to store it in its internal buffers. In the case of large messages (i.e. messages greater than `BIPSMALLSIZE`), a request is sent to the communication thread on the remote node (step 2), asking it to verify the existence of a reception buffer for that message (step 3). This verification is done from a *flag vector*, where there is one entry for each mailbox created in a node. If the response is negative, the sender tries another request. When the receiver executes a retrieve primitive (step 1), the entry related to its mailbox is signaled in the vector. On the next request, the sender receives a positive answer (step 4) and the message can finally be posted to the receiver's mailbox (step 5).

The optimal behaviour to exchange large messages in DECK is described by the relation among steps 1, 2 and 3 in Figure 5: the receiver process should always try to execute the receive primitive before the message is transmitted by the sender process. In this way, when the communication thread fetchs into the vector for a signaled position, it could immediately give a positive answer to the sender, minimizing the time and network traffic used to perform the checking. Concurrent accesses to the flag vector are controlled by means of a semaphore.

IV. Performance evaluation

The performance evaluation of both versions of DECK was made on a cluster of 4 Dual Pentium Pro 200 MHz nodes running Linux 2.2.10. Regarding the DECK/Myrinet implementation, it was adopted the BIP driver 0.99e compiled with SMP support.

The used benchmark was the *ping-pong* algorithm, executed 500 times for each implementation of DECK and considering the mean values for the traditional one-way latency and bandwidth metrics. Time was measured with the `gettimeofday` function, from the C language.

To assess the obtained results, the same bechmark was implemented in PVM and MPI (LAM and MPICH implementations), in the case of Fast Ethernet; and MPI-BIP [WES99] and pure BIP, in the case of Myrinet. Table I shows the set of primitives used for each environment.

TABLE I
COMMUNICATION PRIMITIVES USED FOR PERFORMANCE EVALUATION.

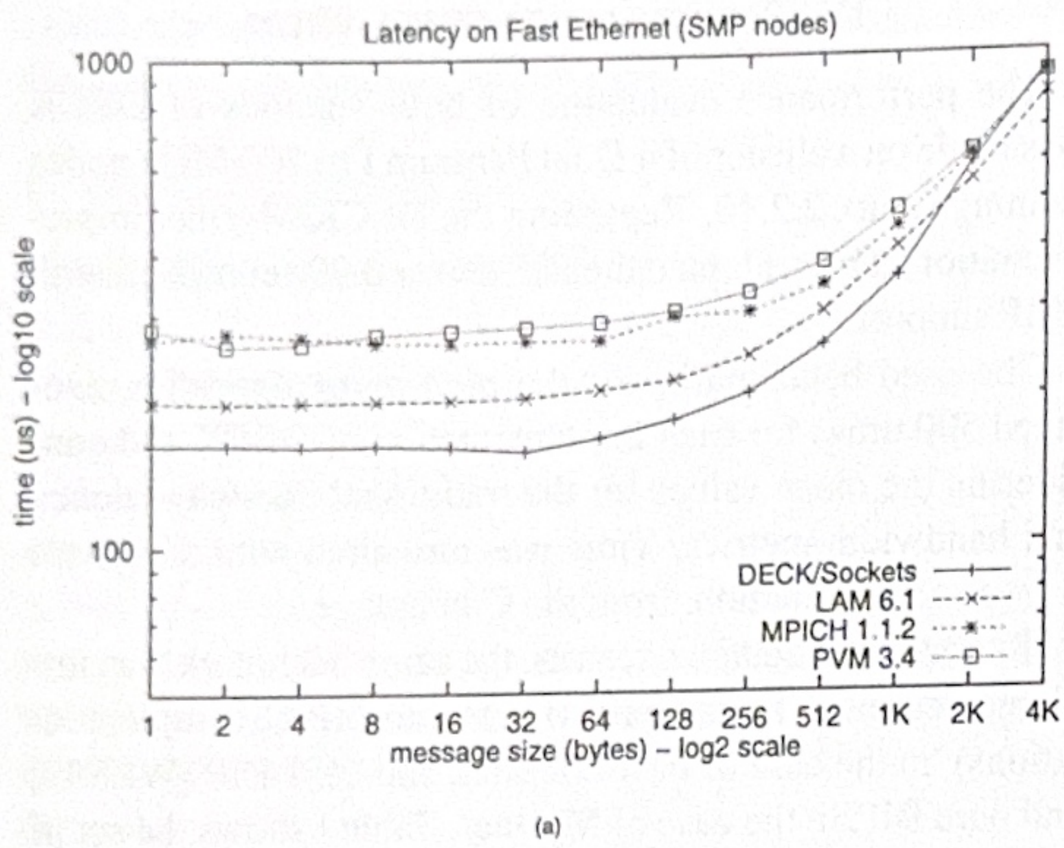
Environment	Primitives
PVM	`pvm_send()`, `pvm_recv()`
MPI	`MPI_Send()`, `MPI_Recv()`
DECK	`deck_mbox_post()`, `deck_mbox_retrv()`
BIP	`bip_tsend()`, `bip_trecv()`

In the evaluation on top of Fast Ethernet, we have used for PVM the flag `PvmDataRaw` and the primitive `pvm_pkbyte()` for data packing, since the underlying architecture is homogeneous and, in this way, no data conversion is needed. Additionally, the option `PvmRouteDirect` with 32 KB of message length was used to establish direct communication between the PVM tasks, avoiding the daemon. For MPI, it was used MPI_BYTE as datatype for send operations. With the LAM 6.1 implementation, the flags `-O` and `-c2c` were used at execution time to indicate that the underlying architecture is homogeneous and to establish direct communication between the processes, with the objective of achieve better performance.

For the Myrinet evaluation, we have use the program *perf.c*, provided within the BIP distribution, to get the results for this protocol; since this program implements the flow-control semantic required by BIP.

A. Fast Ethernet evaluation

Figure 6 depicts the obtained results for latency (*a*) and bandwidth (*b*) of DECK on top of the Fast Ethernet network. For messages greater than 2 KB, the latency for PVM and MPICH implementations is very close to the latency presented by DECK; however, the LAM 6.1 library shows a slightly better performance. It should be noted that DECK's



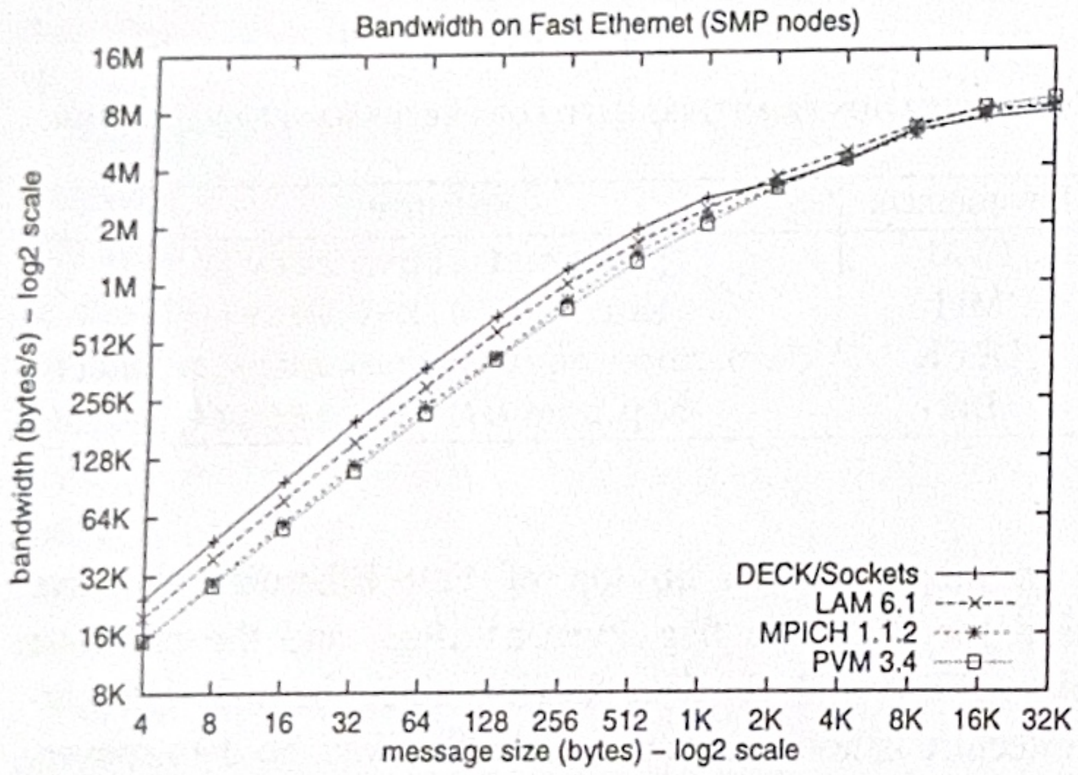


Fig. 6. Latency and bandwidth for Fast Ethernet.

performance is clearly the best one for messages up to 2 KB. For instance, with a message of 1 byte, DECK achieved a latency around 162 μs, against 199 μs for LAM, 267 μs for MPICH and 289 μs for PVM, which represents a speed-up of 19%, 40% and 43%, respectively.

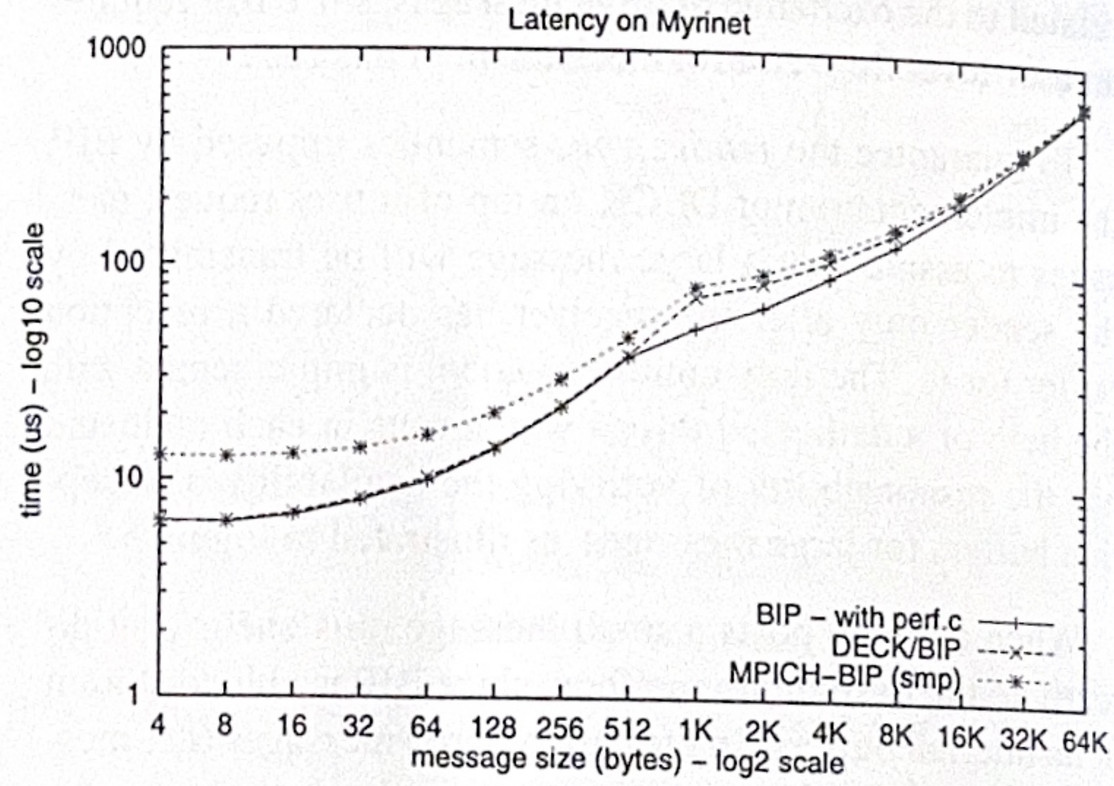
The curves for bandwidth make possible to state that, for small messages — up to 2 KB —, DECK achieves a speed-up of 4%, 10% and 15% in comparison with LAM, MPICH and PVM, respectively. However, for larger messages (starting from 2 KB), all implementations converge to a bandwidth around 8 MB/s, except PVM, which has achieved 9 MB/s.

It is important to say that we have limited the message size in 32 KB for this evaluation due to the datagram sockets used by DECK. As the objective of this first implementation was only validate the structure and functionalities of DECK, we dont expect to use it as a programming environment for complex parallel applications, since we have faster communication networks.

B. The Myrinet evaluation

Figure 7 shows the DECK latency (a) and bandwidth (b) curves in comparison with MPI-BIP and pure BIP.

DECK has achieved a very good performance, showing a latency very close to that of pure BIP, remaining under 10 μs for messages up to 64 bytes. On the other hand, MPICH has presented a great overhead even for small messages — around of 12 μs for 4 bytes messages, whereas DECK has presented around of 6.5 μs for the same message size. These results are probably due to the way MPICH implements its message segmentation strategy.



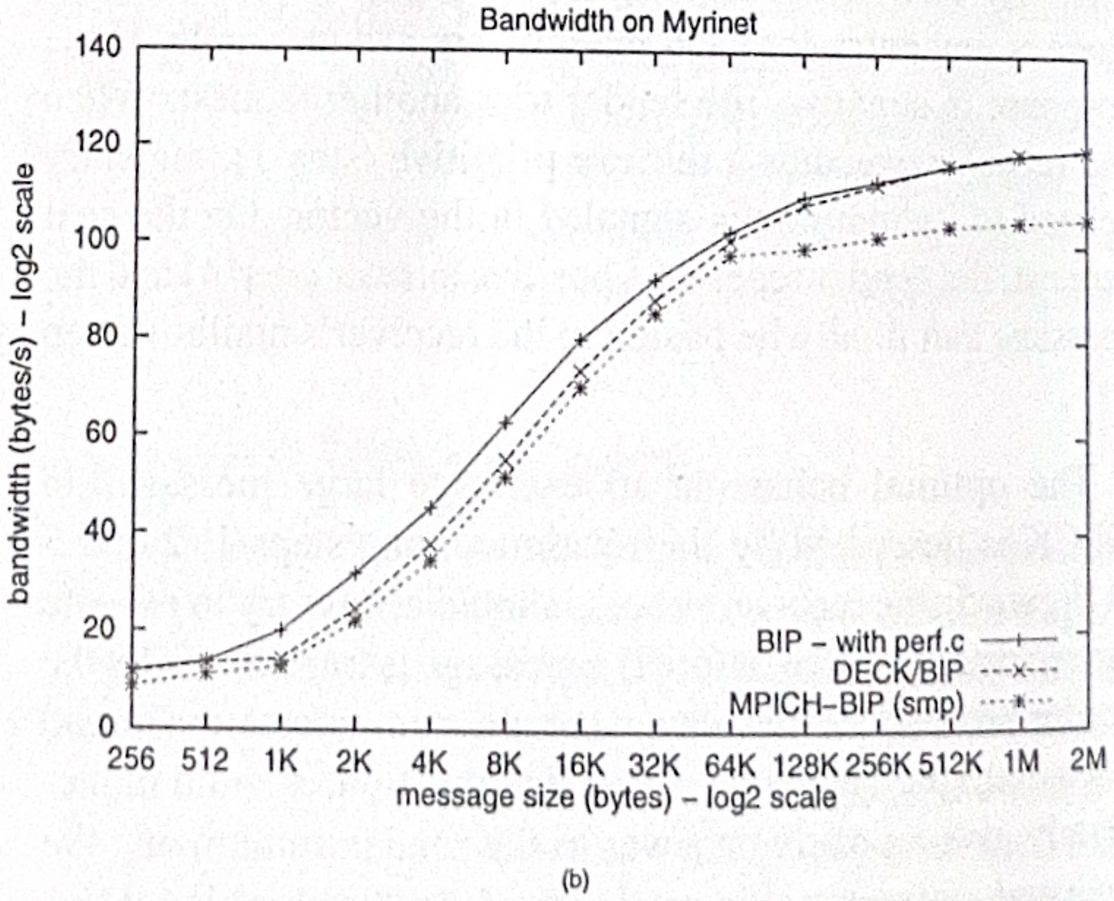


Fig. 7. Latency and bandwidth for Myrinet.

Regarding the bandwidth, it should be noted that for messages greater than 1 KB, the DECK bandwidth decreases in comparison with the BIP one. Not surprisingly, 1 KB is the current threshold between BIP small and large messages; hence, that is the point where DECK starts its flow-control protocol to deal with large messages.

This flow-control protocol has proven to be quite efficient, since DECK is able to achieve a maximum bandwidth very close to that of BIP, respectively 118 MB/s and 120 MB/s. The results for MPICH are also quite satisfactory, remaining near of 106 MB/s. This difference is due to the different implementations of the flow control protocol on both environments: MPICH uses a credit-based protocol, in which there is a number of send and receive tokens that must be passed in each communication operation; DECK simply sends a control message to the receiver process to indicate the size of a message to be transmitted and, after, sends the entire message in only one communication operation. This implementation seems to be better, as we can see through the measured results.

V. Related Work

As mentioned in section I, with the appearance of fast communication networks and, consequently, new communication protocols dedicated to these networks, a great number of parallel and distributed programming environments has arised or adapted to use these new technologies. In this section, we describe some libraries/protocols which are very similar to DECK, in terms of functionalities and objectives.

Regarding MPI implementations, we can highlight the aforementioned MPI-BIP implementation, which offers two distinct semantics to message management: small messages are encapsulated into *control messages*, which have a buffering scheme and are sent directly to the network; large messages are managed by means of a *rendez-vous* protocol, in the same way as DECK; however this implementation is quite efficient, with a small impact on performance (as illustrated in the section IV).

MPI-FM [LAU97] is an implementation of MPI based on the translation of the ADI layer to use FM primitives. This implementation uses internal queues for send/receive operations and message handlers to assure the correct behaviour of MPI applications over the active messages model used by FM. The results for latency (19 μs for zero byte messages) and bandwidth (17.3 MB/s for 2 kB messages, for example) are quite satisfactory.

Athapascan0 is a runtime layer destinated to the support of irregular applications. It is based on the A0kernel, which provides synchronous and asynchronous communication, as well as local and remote thread facilities. Besides, it is used as a basic layer for a distributed dataflow language called Athapascan1. More recently, the development of the Athapascan-SMP [CAR99] has introduced several modifications in the kernel, such as distributed semaphores and scheduling techniques, to better resolve problems related to concurrent accesses to the MPI layer. The functionalities provided by the entire environment are concentrated in the simultaneous support for different communication interfaces within a specific node and on a set of scheduling techniques provided by Athaspascan1.

Nexus is a runtime layer used within the Globus project [FOS98] that provides support for irregular applications. It offers a set of functionalities, such as threads, communication contexts and remote service requests (with an active message model), that could be used to the development of high level programming environments or compilers. Through the Nexus layer, the Globus project intends to furnish a complete programming environment that will be used in grid environments. To achieve this, Nexus also foresees the availability of inter-cluster communication, by means of its global pointers.

PM^2 is a parallel programming environment that provides portability and high performance. It is based on a proprietary threads package, called Marcel threads [MEH95], and on the Madeleine [BOU99] communication layer, which provides synchronous and asynchronous LRPC (*Lightweight RPC*), with an active message model and is responsible to adapt the environment to different communication networks, such as Ethernet, Myrinet and SCI.

DECK differs from these environments in the following points: the DECKs API is quite small and simple, furnishing primitives related to threads and point to point communication. Thread migration and global memory facilities are not currently provided. The upper layer of DECK already provides group communication and load balancing services, which are used on demand and are not currently provided by the other environments. Further, DECK foresees the possibility of supporting different communication interfaces at the same time, as much in the same way Nexus and Madeleine have already provided.

VI. Conclusions and Current Work

In this paper, we have presented the structure and resources provided by DECK, an environment for parallel programming on clusters of multiprocessors.

The modular structure of DECK has proven to be very confortable since we have made two distinct implementations with a small number of modifications in the low-level layer of DECK and without changes in the programming interface.

The first implementation of DECK, based on datagram sockets, presents a medium speed-up around 34% for latency and 10% for bandwidth, considering messages up to 1 KB. Comparing to the results of the remaining environments for the same message size, it is possible to conclude that this

version of DECK is quite favorable, i.e., it provides a true gain to applications that exchange small messages, which are common in typical parallel applications. In this way, for Ethernet-based clusters, this version of DECK could be used as an alternative programming environment instead of PVM and MPI, due to its good results.

The performance results of DECK/Myrinet reveal a good design in terms of latency and bandwidth, having presented a gain of 42% in latency and 12% in bandwidth, when compared to MPI-BIP. Additionally, the flow-control protocol needed to deal with large messages has caused a little impact on the maximum bandwidth achieved by DECK, keeping it very close to BIP.

In short, both DECK implementations are very efficient for small messages and, provided that typical parallel applications frequently exchange this kind of message, DECK can be considered a good choice for parallel programming on Fast-Ethernet or Myrinet-based clusters.

The development of other services is in a design phase. We have already started the discussion on how to efficiently provide fault tolerance for DECK applications, regarding its possible multithreaded behaviour.

Furthermore, due to our MultiCluster project, we are also working on an efficient implementation of DECK on top of SCI network, in order to obtain results close to SCI limits for PC-based platform — 80 MB/s of bandwidth and 2-5 μs of latency. DECK/SCI will be implemented on top of SISCI API [GIA98], which is a low-level programming interface that fully explore the SCI hardware capabilities. After such an implementation, we will be able to carry out the MultiCluster integration model, by interconnecting a Myrinet cluster with a SCI one and providing the DECK users with an uniform programming interface despite the hardware heterogeneity, always keeping track of the underlying high-performance networks. The MultiCluster integration is supported by DECK through the RCD mechanism, mentioned in the section that pointed out the DECK software structure.

ACKNOWLEDGEMENTS

This work has been partially supported by grants from CNPq, CAPES, FINEP and UFRGS. The authors would like to thank Patrick Geoffray and Roland Westrelin, from the BIP team, for their help in installing and using the BIP protocol.

REFERENCES

[ÁVI99] ÁVILA, Rafael Bohrer. **Um modelo de paralelismo de grão fino para objetos distribuídos**. Porto Alegre: PPGC/UFRGS, 1999. Dissertação de Mestrado.

[BAR00] BARRETO, Marcos; ÁVILA, Rafael; NAVAUX, Philippe. The MultiCluster model to the integrated use of multiple workstation clusters. In: WORKSHOP ON PERSONAL COMPUTER BASED NETWORKS OF WORKSTATIONS, 3., 2000, Cancun. **Proceedings...** Berlin: Springer, 2000. p.71–80. (Lecture Notes in Computer Science, v.1800).

[BAR98] BARRETO, Marcos E.; NAVAUX, Philippe O. A.; RIVIÈRE, Michel P. DECK: a new model for a distributed executive kernel integrating communication and multithreading for support of distributed object oriented application with fault tolerance support. In: CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN, 4., 1998, Neuquén, AR. **Anales...** Neuquén: Universidad Nacional de Comahue, Facultad de Economia y Administración, Departamento de Informática y Estadística, 1998. v.2, p.623–637.

[BOD95] BODEN, N. et al. Myrinet: a gigabit-per-second local-area network. **IEEE Micro**, Los Alamitos, v.15, n.1, p.29–36, Feb. 1995.

[BOU99] BOUGÉ, Luc; MEHAUT, Jean-François; NAMYST, Raymond. **MADELEINE**: an efficient and portable communication interface for RPC-based multithreaded environments. [S.l.]: INRIA - Institut National de Recherche en Informatique et en Automatique, 1999. (RR 3459).

[CAR98] CARISSIMI, Alexandre; PASIN, Marcelo. Athapascan: An experience on mixing MPI communications and threads. In: IPPS/SPDP'98 WORKSHOPS, 10., 1998. **Proceedings...** Springer, 1998. p.137–144. (Lecture Notes in Computer Science, v.1388).

[CAR99] CARISSIMI, Alexandre. **Athapascan-0**: exploitation de la multiprogrammation légère sur grappes de multiprocesseurs. Grenoble, FR: Institut National Polytechnique de Grenoble (INPG), France, 1999. Tese de Doutorado.

[CAS00] CASSALI, Ricardo et al. Group communication service for DECK. In: , 2000. **Anais...** 2000. (To be presented at CLEI 2000: Conferencia Latinoamericana de Estudios en Informática - Atizapán de Zaragoza, Mexico (sep./2000)).

[CUN99] CUNNINGHAM, David; LANE, William. **Gigabit Ethernet networking**. [S.l.]: Macmillan Technical Publishing, 1999.

[FOS98] FOSTER, I.; KESSELMAN, C. **Globus**: a metacomputing infrastructure toolkit. WWW, dez. 1998.

[FOS94] FOSTER, I.; TUECKE, S.; KESSELMAN, C. Nexus: runtime support for task-parallel programming languages. In: INTERNATIONAL WORKSHOP ON PARALLEL PROCESSING, 1994. **Proceedings...** New Jersey: McGraw-Hill, 1994.

[FOS96] FOSTER, I.; TUECKE, S.; KESSELMAN, C. The Nexus approach to integrating multithreading and communication. In: PARALLEL PROGRAMMING ENVIRONMENTS FOR HIGH PERFORMANCE COMPUTING, 1996, Alpes d'Huez, FR. **Anais...** 1996. p.53–67.

[GEI94] GEIST, Al et al. **PVM**: parallel virtual machine. Cambridge, MA: MIT Press, 1994.

[GIA98] GIACOMINI, F. et al. **Low-level SCI software requirements, analysis and predesign**. [S.l.]: ESPRIT Project 23174 — Software Infrastructure for SCI (SISCI), 1998.

[GIN97] GINZBURG, I. **Athapascan0b**: intégration efficace et portable de multiprogrammation légère et de communication. Grenoble: Institut National Polytechnique de Grenoble (INPG), 1997. Tese de Doutorado.

[GM99] GM. Available by WWW at http://www.myri.com/GM (dez. 1999), 1999.

[IEE92] INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS. **IEEE standard for scalable coherent interface (SCI)**. IEEE 1596-1992, 1992.

[IEE95] INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS. **Local and metropolitan area networks-supplement—media access control (MAC) parameters, physical layer, medium attachment units and repeater for 100Mb/s operation, type 100BASE-T (clauses 21–30)**. IEEE 802.3u-1995, 1995.

[IEE95a] INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS. **Information technology—portable operating system interface (POSIX), threads extension [C language]**. IEEE 1003.1c-1995, 1995.

[IPPS98] IPPS/SPDP'98 WORKSHOPS, 10., 1998. **Proceedings...** Springer, 1998. (Lecture Notes in Computer Science, v.1388).

[LAU97] LAURIA, Mario; CHIEN, Andrew. MPI-FM: high performance MPI on workstation clusters. **Journal of Parallel and Distributed Computing**, Orlando, FL, v.40, n.1, p.4–18, Jan. 1997.

[MEH95] MEHAUT, Jean-François; NAMYST, Raymond. **Marcel**: une bibliothèque de processus légèrs. [S.l.]: Laboratoire d'Informatique Fondamentale de Lille, 1995.

[MPI94] MPI FORUM. **The MPI message passing interface standard**. Knoxville: University of Tennessee, 1994.

[NAM96] NAMYST, Raymond. **PM²**: un environnement pour une conception portable et une execution efficace des applications parallèles irrégulières. Lille, FR: LIFL, Université de Lille, 1996. Tese de Doutorado.

[PAK96] PAKIN, S.; LAURIA, M.; CHIEN, A. High performance messaging on workstations: Illinois Fast Messages for Myrinet. In: SUPERCOMPUTING'95, 1996, San Diego, CA. **Proceedings...** IEEE Computer Society Press, 1996.

[PRY98] PRYLLI, Loïc; TOURANCHEAU, Bernard. BIP: a new protocol designed for high performance networking on Myrinet. In: IPPS/SPDP'98 WORKSHOPS, 10., 1998. **Proceedings...** Springer, 1998. p.472–485. (Lecture Notes in Computer Science, v.1388).

[VIA99] VIA — Virtual Interface Architecture. Available by WWW at http://www.via.org (dez. 1999), 1999.

[WES99] WESTRELIN, Roland. Une implémentation de MPI pour réseaux locaux a très haut débit: MPI-BIP. In: RENCONTRES FRANCOPHONES DU PARALLÉLISME, 11., 1999, Rennes. **Proceedings...** Lyon: INRIA, 1999.