# CoR's Faster Route over Myrinet

António Pina, Albano Alves, Vítor Oliveira, Cecília Moreira
Grupo de Engenharia de Computadores
Dep. Informática, Universidade do Minho
Campus de Gualtar, Braga, Portugal
{pina,albano,vspo,ceci}@gec.di.uminho.pt

## Abstract

*In this paper we concentrate in the efforts made to exploit the performance of Myrinet to build a faster communication route into CoR*[1]*. By accessing the Myrinet interface through GM*[2]*, we achieved low latency and high bandwidth message passing without the overhead of a higher level protocol stack, system calls or interrupts. CoR is an ongoing project unique in its design goal of combining multithreading, message passing and distributed shared memory with facilities to dynamically select from different transport media and protocols the one that best fits communication and interaction requirements. The ability to mix CoR and PVM calls in the same program brings numerous benefits to the application developer familiar with PVM, notably: 1) new transport communication layers; PvmRouteMyrinet and PvmRouteUdp; 2) migration mechanisms for exploiting fine grain message passing; 3) thread-safe communication PVM API; 4) object-oriented distributed shared memory.*
**Keywords***: clustering, multithreading, message routes.*

## 1. Introduction

Application areas such as computer graphics and multimedia, information systems, decision support and transaction processing are likely to see a tremendous transformation as a result of the vast computer power available at low cost through parallel computing.
The overhead of initiating and receiving communication is greatly influenced by the extent to which the necessary tasks can be performed by hardware rather than being delegated to software, particularly the operating system. The technology breakthrough that represents the potential of clusters taking on an important role in the large scale parallel computing is a scalable low-latency interconnect, similar in quality to that available in parallel machines, but deployed like a local-area network.
A major influence on clusters has been the rise of popular public domain software, such as PVM[7] and MPI[6], that allow users to farm jobs over collection of machines or to run a parallel program on a number of machines connected by an arbitrary local-area or even wide-area network. Although the communication performance capability is quite small, typical latencies are 1 millisecond or more for even small transfers and the aggregate bandwidth is often less than 1 Mbytes, these tools provide an inexpensive vehicle for a class of problems with a very high ratio of computation to communication.
As distributed computing has evolved, it has come to evidence that no one monolithic system can handle efficiently all the desired communication styles. We have also learnt that to better manage flexibility and interoperability and achieve critical performance the semantics of a particular message style must be decoupled from the low-level infrastructure.
Most distributed applications use TCP/IP suite of protocols because they are well understood and widely available. But they focus more on data integrity and sequenced delivery rather than low latency, which makes them less appropriated for distributed applications since low latency and small messages transferring bandwidth are major concerns here.
The high speed networking technology developed for large scale parallel machines has migrated down into a number of widely used local area networks (LANs). In addition, a number of higher bandwidth, lower latency system based networks are becoming to be known as *system area network* (SAN). As opposed to tightly packaged parallel machine network or widely dispersed local area network, SAN networks have been commercialised, such as Myrinet[4] and SCI[5], which operate over shorter physical distances.

[1]Resource oriented Computing
[2]A message-based communication system for Myrinet

## 2 Overview of CoR

CoR is an evolving project aimed to translate the vast computer power available at low cost through multi-threaded parallel computing into greater performance and expanded capability. It has been primarily motivated by the need of creating a parallel computer environment to support the design and evaluation of applications that conforms with the $MC^2$ (Cellular Computation Model)[11]. The model combines production systems with Petri Nets as a way of specifying and regulating the overall activity of a distributed system viewed as a multi-cellular agent[12]. The final goal is to investigate and construct a common framework to understand and evaluate architectural and logical trade-off, making it possible to undertake quantitative as well as qualitative studies of software/hardware interaction.
CoR introduces the **resource** as its sole metaphor that shifts the traditional boundaries between hardware (physical resources) and software (logical resources) to make programming task easier and performance more robust. The specification, coordination and execution of applications lay on the definition of a variety of physical and logical resources, such as domains, tasks, data, ports, transitions, synchronizers, barriers, topologies, etc.
CoR can be viewed as an extension to the process abstraction that uses **domains**, **tasks**, **ports** and **data** basic resources to achieve structured fine-grained computation and communication with control and object-oriented distributed shared memory. A task is an execution resource while domains delimit regions of addressable space where interaction between resources occurs. Ports are communication abstraction mechanisms used to route the information between domains. Data resources are a general class used to deal with a great variety of different types of structured and unstructured pieces of information.

### 2.1 Domains

In a system of distributed domains, every resource is defined by a **body** comprising various elements and an **idp** which is an application universal identifier. This idp is sub-divided into a *macro-identification* and a *micro-identification*; the last being used to uniquely identify the resource of a distributed application, whereas the first encode an index being used to join and integrate on-the-fly pluggable resources, belonging to different applications. As resources always exist within the context of domains, the system also provides for a local identification – a string *name* and an *index* relative to the resource **ascendant** domain which is also a resource owning an universal identifier. The programmer can arbitrarily use either local or universal identifiers to designate a resource, however the semantic of the operation is most of the times identifier dependent, as it is the case with the communication operations.
The local naming facilities offered by domains are extended by *logical domains* – that only serve as "shells" or environments to other resources to come into existence. A logical domain is simply a set of resources attached to a particular idp with an order and name provided on this set. Several existing systems use groups (the equivalent of a logical domain) for slightly different reasons[3, 6], allowing either static or dynamic membership allocation.

### 2.2 PVM interoperability

As a first effort to get CoR running we make it compatible with PVM, taking advantage of existent commodity hardware/software communication layers.
The approach brings enormous benefits to users. First, it provides a migration mechanism for application developers familiars with PVM. Existing programs can be ported in stages to CoR as the developer learns more about CoR functionality.
Secondly, by mixing CoR and PVM calls in the same program the platform brings numerous benefits to the application developer familiar with PVM, notably: 1) new transport communication layers; PvmRouteMyrinet and PvmRouteUdp; 2) migration mechanisms for exploiting fine grain message passing; 3) thread-safe communication PVM API; 4) object-oriented distributed shared memory.

## 3 CoR system architecture

CoR's environment prototype is based on a former effort to combine multithreading, message passing and shared memory to: 1) exploit small scale SMP, 2) refine the grain of parallelism and 3) overlap computation and communication on a single processor[10].
CoR overcomes several disadvantages of the process oriented model by offering smaller granularity and lower task initialisation and scheduling costs. Support of multiple resource representations schemes eases the transition from a concurrent programming model using shared memory to a distributed resource-oriented run-time system with support to message passing. On each domain running on a uniprocessor or a scalable SMP, a *domain core* relying on POSIX threads schedules both domain threads and user-tasks.
The core consists of several independent subsystems each one dedicated to a different service realized in hierarchy layers from a low-level interface of system dependent services – POSIX threads, dynamic libraries, PVM and other network transport communication agents – to the higher level interfaces of the system and application APIs (see figure 1). It supports both local and remote operations

among resources that spawns the entire parallel machine offering the potential for fine-tuning computing and overlapping of communication and computation.

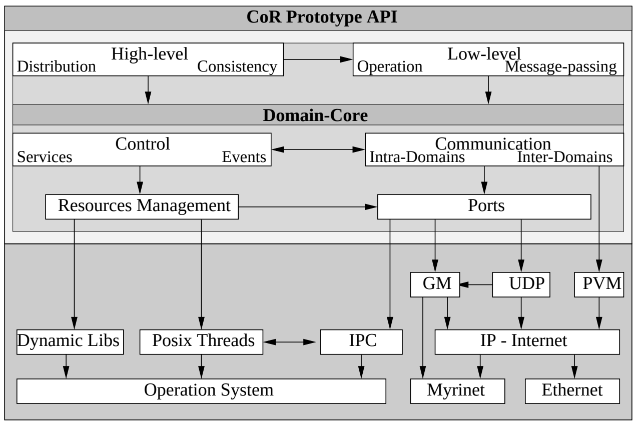


**Figure 1. Domain layers hierarchy.**

### 3.1 Programming model

CoR is an emerging paradigm that simplifies programming description and execution by freeing the user from the burden of explicitly managing the complex relationships between the two phases, typically, involved on the development of a new application program – *structuring* and *computing*.
By structuring we mean the definition of a variety of logical/physical structures and the naming of the entities used by the application. Computing is related to a continuous process of inquiring, transforming and communicating the state of the named entities by executing the instruction of a distributed multi-threaded control parallel program.
Logical domains are used to support the design of large, complex and modular applications, by organizing resources in a hierarchical tree of dependencies, where nodes are *structured resources* (domains) and leaves *simple resources*.
CoR communication model assumes that any task can send a message to any other task within any domain in the parallel machine. Each message is labelled with a user supplied tag before sending. The receiver task matches on this tag (and on the source of the message), thus allowing to discriminate between multiple messages arriving at the same time. In order to generalize it is also important to allow the use of parallel libraries in conjunction with application message-passing; however the communication required by the library must be isolated from other application communication.
The support for parallel libraries is broadly similar to that of MPI[6] where messages are addressed through a domain resource which can be viewed as a bundle of a message context and a resource group. The defining property of a context is that a message cannot be received in a context other than within which it was sent. CoR assigns different message contexts and resource groups to each domain in the parallel machine, providing a safe space for library writer and user alike.
The idp of a domain resource can be used along with the *user-tag* arbitrarily assigned by the programmer as a *context-tag* assigned by the system, which has the effect of dividing messages into separate communication spaces. The sender (receiver) of a message must always be specified as a parameter of the communication operation call. When the domain is used as a context-tag the source (destination) of a message must be specified as a local identifier (to the domain). In alternative the source (destination) may be supplied as an universal identifier.

## 4 Fast message passing

CoR is designed to support a multi-user parallel processing environment based on a message passing paradigm which guarantees that messages originated from the same path are received in the order they are sent, without loss.
In each node of the parallel machine each communication hardware interface is assumed to be partitioned by the underlying system into a certain number of disjoint partitions called **ports** that are hardware and software dependent communication routes. Each port is a connectionless communication path to other hardware compatible ports used to communicate with other domains in the parallel machine.
At run-time, inside each domain, to fully abstract from the specific communication medium and protocols, several *soft-ports* can be open to share or exclusively use the underlying communication network hardware, and then closed releasing the ports. In figure 2 we show several domains each one having two ports sharing the available interfaces in the respective node.

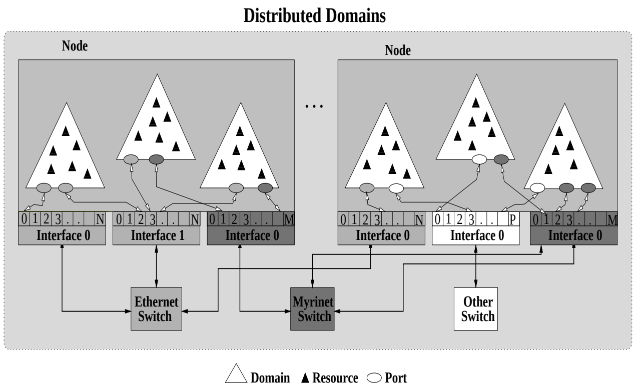


**Figure 2. CoR parallel machine.**

### 4.1 Multiple communication ports

In most cases an application needs not to be aware of the number and type of ports used because the library attempts to optimise ports utilization to prevent the loss of concurrency and bandwidth. However, this may not be the optimal situation when there are some aspects of the communication that are required to be satisfied, such as the domain's total aggregated bandwidth, or fast communication requirements. In those situations soft-ports may be explicitly bounded to specific communication port interfaces and optionally selected the communication protocol.
A conventional domain has a single soft-port bounded to a hardware communication interface appropriated for most applications. The reasons for supporting multiple-ports fall into two categories: those motivated by the need of isolating communication between different communication paths and those motivated by application concurrency. In the first category, user tasks and system threads may link to pre-assigned ports on a per-domain base to get high-bandwidth or low-latency communication.
The second reason for multiple-ports is application requirements of concurrency. Many applications are best structured as several independent tasks that keep moving from request to request to satisfy many resource interactions in progress. In this case the existence of multiple soft-ports bounded to an equivalent number of ports or schedule by the available ports may be the most suitable solution to ensure the concurrency and the responsiveness of the system.

### 4.2 New message routes

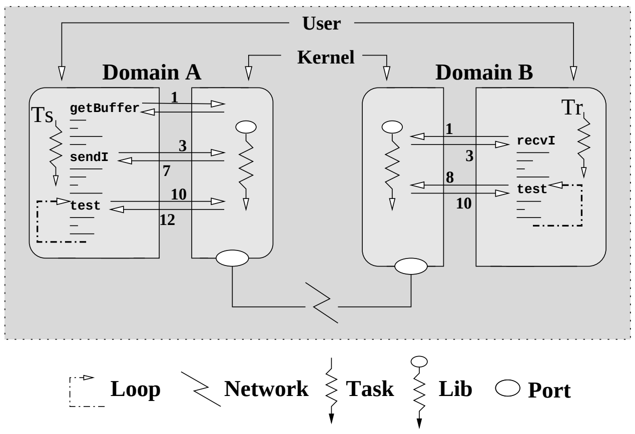
In our laboratory we are running a low-cost high-performance parallel machine using PC commodity hardware. We decided to use Myrinet, which is not a commodity hardware product, because low CPU overhead, low latency and high bandwidth are fundamental concerns on distributed parallel computing.
Considering the technical difficulties to reach raw performance using directly the lower layers of Myrinet hardware, we chose to build the route using a low message passing layer. There are several different implementations of message passing layers on Myrinet such as GM[8, 2], FM[9] and PM[13].
GM provides reliable ordered delivered between communication endpoints, called "ports". The model is connectionless however GM maintain reliable connections between each pair of nodes in the network by multiplexing the traffic between ports over those reliable connections. Since it is possible to obtain TCP/IP running over GM, the current version of CoR based on PVM can run over Myrinet. However, due to the expensive protocol overhead, the result is inefficient parallel execution.
As a first tentative to study scalability we built another basic route based on UDP (on top of IP Ethernet and GM) that runs independently from other TCP/IP socket parts of the CoR environment over the two networks.
Following the same approach used to write the UDP-oriented communication layer, we created a message route directly over GM to bypass the TCP/IP protocol stack.
Finally, by comparing and refining those two communication layers we produced a common message passing interface specification and rewrote the two final – *CoRudpRoute* and *CoRgmRoute* – portable run-time pluggable communication modules[3].
CoR allows new and existing PVM application programs to benefit from the new message routes by extending the PVM API call *pvm_setopt* with two new options: *PvmRouteDirectUDP* and *PvmRouteDirectMyrinet*, without making any modification to the underlying communication subsystem.

### 4.3 Core domain communication

In CoR every domain in the parallel machine has the potential to perform local and remote communication supported by a core communication layer in a per-domain base. The maximum number of ports available on each domain is communication interface dependent, as is the mapping strategy used to assign soft-port to hard-ports. For each existent connection to a hard port (see figure 2) the domain creates a certain number of communication servers (core threads), whose purpose is twofold: 1) **listen** for incoming messages, storing the data in message buffers previously allocated and signal the reception of the message, 2) process and **dispatch** the messages coming from the user layer to a remote domain signalling the termination of the operation.
CoR provides several basic primitives to handle task-to-task communication (see figure 3). This set of primitives assumes an asynchronous communication model; any send or receive operation returns immediately a handle. Later, the handle may be used for test or wait for operation completion. We assume a general communication module where send and receive buffers of acceptable size must reside in the communication core, explicitly allocated by a parameterised call to the library.

### 4.4 Communication protocol

The special case of point to point resource interaction occurs by means of standard communication operations called by tasks when the source and the destination tasks reside in disjoint domains.
As depicted in the figures 4 and 3, **Ts** and **Tr** are, respectively, the sender and the receiver tasks that we will use to explain the general interaction mechanism used by resources. The numbers in parentheses are the same used in

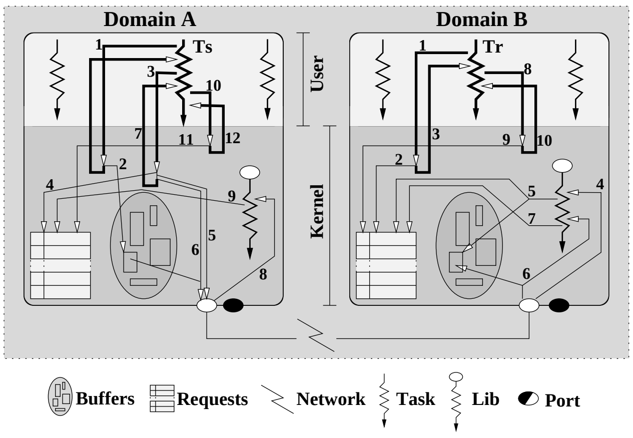


| Sending | Receiving |
|---|---|

```
setTransport(Myrinet);            hdl = recvI(ctx, A, Ts,
ptr = getBuffer(type, n);             tag, type, n, &ptr);
[...]                             [...]
hdl = sendI(ctx, B, Tr,           while(!test(hdl)){
      tag, type, n, ptr);         [...]
[...]                             }
while(!test(hdl)){
[...]
}
```

**Figure 3. Task-to-task communication.**

the figures 4 and 3 for the sending and the receiving domains. An asynchronous sending operation comprises three steps in CoR. First the sender Ts must allocate (*1*) a send buffer, second the buffer must be filled, third the complete message (user tag, context, data type ... ) is sent (*3*) to the receive task. As we are dealing with an immediate operation the sender receives a handle that can, later, be used to probe (*10*) for the completion of the initiated operation.
At the other side of the operation, the receiver task Tr, in a remote domain, executes an immediate receive (*1*) operation set to accept a specific message from task Ts. As above, it receives a handle that may be used latter in the same way (*8*) as the sending operation call.
In the communication core of the domain of Ts, the library allocates (*2*) the requested buffer internally registering (*4*) in the request queue the Ts sending (*3*). Every sent message is divided in two: a control message (*5*) requesting a buffer on the peer communication server of the peer domain and a data message containing the data to be sent in (*6*). Finally a handle that uniquely identifies the request is returned to Ts (*7*). Step (*8*) produces a status code – the combined result of (*5*) and (*6*) – used as the code completion of the operation in (*9*) to update the request queue. Finally, when Ts inquires the status of the sending operation (*10*), using the respective handle, the library dispatcher probes the operation (*11*) returning to (*12*) the code produced on step (*8*).
In the communication core of the domain of Tr the library dispatches the receive call as a conventional asynchronous operation by internally registering it in the request queue (*2*) and returning to Tr (*3*) a handle that uniquely identify the pending request. As a response to the send control coming from the peer domain the interface communication hardware asynchronously activate the library listener (*4*) that manages to allocate the requested buffer (*5*). It also receives the data (*6*) and registers the completion status in the request queue (*7*). Steps (*9*) and (*10)* repeat the equivalent steps in the sending node but in this case applied to a receive asynchronous operation.



| Sending | Receiving |
|---|---|
| **1- getBuffer** | **1- receiveI** |
| 2- buffer allocation | 2- regist operation |
| **3- sendI** | 3- return handle |
| 4- regist operation | 4- receive (control) |
| 5- send control | 5- buffer allocation |
| 6- send data | 6- receive (data) |
| 7- return handle | 7- ok (operation) |
| 8- ok (control + data) | **8- test** |
| 9- regist op. end | 9- probe (handle) |
| **10- test** | 10- return status |
| 11- probe (handle) | |
| 12- return status | |

**Figure 4. Detailed view of task-to-task communication.**

## 5 Evaluation

To conduct a set of general evaluation performance tests, we used a PC cluster comprising eight Intel Pentium II 350Mhz dual-processor nodes with 128 MB RAM, running RedHat 6.1 Linux with kernel 2.2.15 SMP. The network infrastructure was based on both Fast Ethernet and Myrinet network interfaces. The Myrinet interface used is based on LANai 4.2, PCI 32bit, 33Mhz.
The tests measured a message passing operation between two selected nodes and domains, after executing a large

number of iterative sequences, not taking into account program setup nor the time to prepare/process sending/receive data. All communications occurs at the user level; messages go from user address space to user address space, as we are more concerned on higher-level communication facilities.

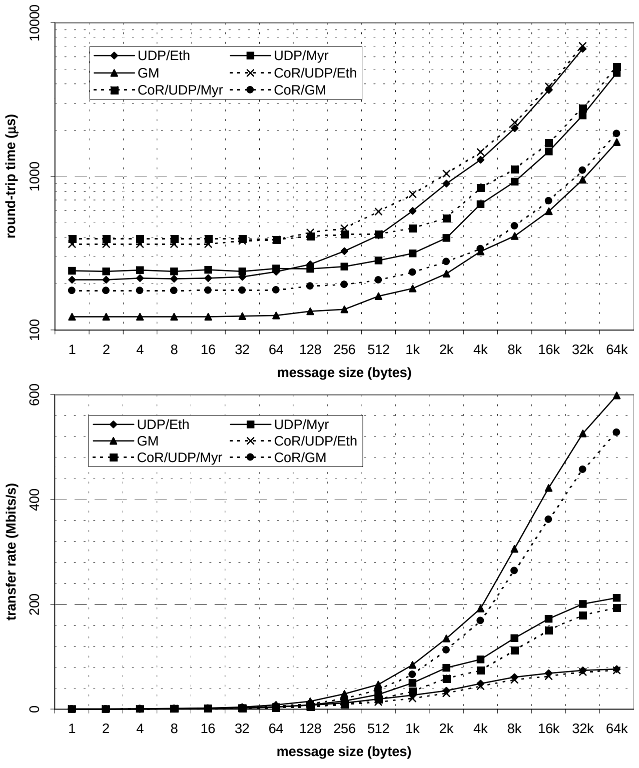


**Figure 5. Round-trip time and transfer rate comparison.**

We performed both round-trip and transfer rate tests. The transfer rate was not calculated based on round-trip tests (ping-pong bandwidth) because we were interested on one-way communication. This avoids the overlap of some messages due to full-duplex capacity when using several communication entities on each node. The overhead of the underlying communication subsystem due to the use of more than one communication pair and more than one port of the communication interface was also measured.

## 5.1 Raw performance

We started by comparing the raw performance of message passing between the different message routes. The message round-trip and transfer occurs at the kernel communication core between pairs of system threads located across the two domains. We exclude the overhead at the domain user communication layer level; this corresponds to basic distributed programming, where each communication entity uses only one port.

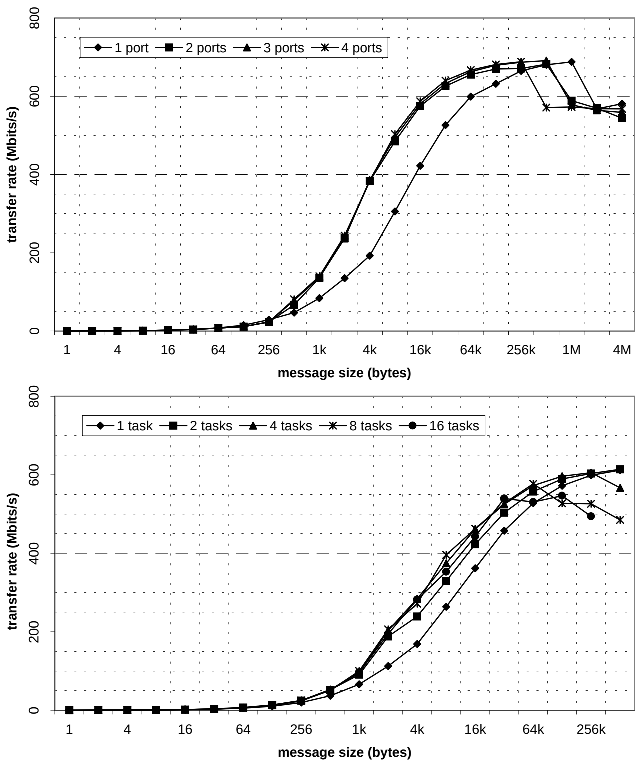


**Figure 6. Transfer rate for CoR/GM.**

Starting with UDP, for UDP over Ethernet IP in figure 5 (*top*) we notice an approximate 212$\mu$s small messages round-trip, while for UDP over GM IP round-trip times almost reduces to half on longer messages. Using GM blocking receives, a round-trip time above 122$\mu$s was obtained[3]. Comparing transfer rates in figure 5 (*bottom*), it can be seen that the peak values are 80 Mbit/s (16kbytes messages), 210 Mbit/s and 600 Mbit/s (64kbytes messages) for UDP over Ethernet IP, UDP over GM IP and GM, respectively. It is important to note that performance is highly dependent on message size and Myrinet's bandwidth can only be efficiently exploited using longer messages.

In figure 6 (*top*) we present transfer rate values for 1, 2, 3 and 4 ports utilization in GM (one thread for each port). We intended to evaluate the overhead when using several communication pairs and ports. This is relevant because a high number of distributed applications use several communication entities, thus requiring different GM ports. As we can see in figure 6, the transfer rate increases with the number of ports and doesn't degrade significantly with the number of tasks. With two ports the graphic presents the total transfer

[3]Using th `gm_allsize` utility provided by GM package, we obtained latency values above 23$\mu$s for non-blocking receives.

rate obtained with both threads.

### 5.2 Task communication

Next we evaluated the two new communication routes under CoR using the same parameters as the raw performance tests, measuring the influence of multi-task competition both at the domain level and at the machine level. Later we decided not to take into account the machine level, because there is not a significant difference between the two levels with the Linux implementation of Posix threads.
The domains contained the same number of send/receive user-tasks, which varied between experiments. Each CoR domain also contains the system thread responsible for handling all the port-to-port communications: listening for incoming data containing the messages received from the remote domain and dispatching the sending request with origin on the user(s) task(s).
Each user-task is engaged on communicating with a paired user-task within a remote domain. Considering that for each pair of user-tasks one is select as the initial sender and the other is selected as the initial receiver, the communication proceeds as a loop that repeatedly executes the following steps: **1**) each pair of user-tasks activate the dispatcher thread by entering an asynchronous send message passing call; **2**) the listener thread on the receiver domain processes the incoming messages and delivers them to the receiver user-task, who has already called an asynchronous receive operation; **3** ) each pair of user tasks exchange the initial role; **4**) re-enters the loop.
In figure 5 we show that CoR overhead at the domain user level of the communication layer is minimum. In this experiment we used two domains and one task per domain.
In figure 6 (*bottom*) we present the overhead of CoR library when using several tasks per domain. It is important to notice that with 2 or 16 tasks we achieve almost the same transfer rate, which doesn't degrade with the increase of the number of tasks.

## 6 Conclusions

PC clusters using high-speed network are very cost effective but, nevertheless, are able to achieve a performance comparable to that of expensive massively parallel machines. A multi-user distributed parallel programming environment can be realized using software such as PVM and MPI on top of TCP/IP suite of protocols. However, this expensive protocol overhead results is inefficient parallel execution.
The CoR prototype provides a robust environment and base for future extension of its distributed multithread-oriented run-time system. In this paper we present our efforts to enhance CoR with new scalable and faster message passing routes over Myrinet network[1].
The first effort based on UDP protocols lead to the developing of a scalable and portable direct message route on top of IP. IP is also implemented by Myrinet API enabling the new route to be used both with Fast Ethernet and Myrinet. By comparing the performance of point-to-point message passing obtained on both networks we notice a significant improvement when using Myrinet. The second effort using GM – a low-level message-based communication system for Myrinet – conduced to the built of a much faster message passing route than the one obtained with UDP.

## References

[1] CoR World Wide Web Page. http://www.gec.di.uminho.pt/.

[2] Myrinet world wide web page. http://www.myri.com/.

[3] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Recent Enhancements to PVM. *International Journal of Supercomputing Applications and High Performance Computing*, 1995.

[4] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–38, February 1995.

[5] M. Fischer and J. Simon. Embedding SCI into PVM. In *Recent Advances in Prallel Virtual Machine and Message Passing Interface*, volume 1332 of *Lecture Notes in Computer Science*. Springer, November 1997.

[6] M. Forum. MPI: A Message-Passing Interface Standard. *Internacional Journal of Supercomputer Application*, 8(3/4):165–416, 1994.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A Users Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation. MIT Pres, 1994.

[8] Myricom. *The GM Message Passing System*, 1999.

[9] S. Pakin, M. Lauria, and A. Chein. High performance messaging on workstations: Illianois fast messages (fm) for myrinet. In *Proceedings of Supercomputing '95 San Diego California*, 1995.

[10] A. Pina, V. Oliveira, and C. Moreira. Domains, Threads and Shared Memory in a message passing environment. Technical report, Departamento de Informática, Universidade do Minho, Braga, Portugal, May 1997.

[11] A. M. Pina. $MC^2$ *– Modelo de Computação Celular. Origem e Evolução*. PhD thesis, Departamento de Informática, Universidade do Minho, Braga, Portugal, 1997.

[12] A. M. Pina, J. M. Fernandes, and R. J. Machado. Genetic regulatory mechanisms by means of Extended Interactive Petri Nets. In *IEEE International Conference on Systems, Man, and Cybernetics (SMC'97)*, Hyatt Orlando, Orlando, Florida, USA,, 1997.

[13] H. Tezuka, A. Hori, and Y. Ishikawa. Pm: A high performance communication library for multi-user parallel environments. Technical report, Tsukuba Research Center, RWCP, Japan, 1996.