# Scalable Multithreading in a Low Latency Myrinet Cluster[⋆]

[1] Instituto Politécnico de Bragança
albano@ipb.pt
[2] Universidade do Minho
pina@di.uminho.pt

**Abstract.** In this paper we present some implementation details of a programming model – pCoR – that combines primitives to launch remote processes and threads with communication over Myrinet. Basically, we present the efforts we have made to achieve high performance communication among threads of parallel/distributed applications. The expected advantages of multiple threads launched across a low latency cluster of SMP workstations are emphasized with a graphical application that manages huge maps consisting of several JPEG images.

## 1 Introduction

Cluster computing is a new concept that is emerging with the new advances in communication technologies; several affordable heterogeneous computers may be interconnected through high performance links like Myrinet.

Using these new computing platforms several complex problems, which in the past have required expensive mainframes, may now be solved using low cost equipment. Particularly, we are interested in providing cluster solutions for informational problems that require a combination of massive storage and moderate computing power.

### 1.1 Resource Oriented Computing – CoR

The CoR computing model has been primarily motivated by the need of creating a parallel computer environment to support the design and evaluation of applications conforming to the MC$^2$ (Cellular Computation Model) [17].

A full specification of CoR and an initial prototype – pCoR – were presented in [14] and [18]. CoR paradigm extends the process abstraction to achieve structured fine-grained computing using a combination of message passing, shared memory and POSIX threads. Specification, coordination and execution of applications lie on the definition of a variety of physical and logical resources, such as domains, tasks, data, ports, synchronizers, barriers, topologies, etc.

[⋆] Research supported by FCT/MCT, Portugal, contract POSI/CHS/41739/2001, under the name "SIRe – Scalable Information Retrieval Environment".

First attempts to introduce high performance communication into CoR, exploiting Myrinet, were presented in [1]. Preliminary results and validation were obtained with the development of a distributed hash system [19] and a global file system to exploit a local Myrinet cluster particularly for information retrieval.

### 1.2 Multithreading and Message Passing

Multithreading and message passing are two fundamental low-level approaches to express parallelism in programs. The first approach proved to be convenient in SMP workstations and the latter is widely used to program applications that distribute computations across networked machines.

Considering that most clusters are built from multiprocessor machines, there is a strong motivation to use a hybrid approach, combining multithreading, shared memory and message passing. This is not an easy task because message-passing primitives of most communication libraries are not thread safe. For instance, the device driver to interface Myrinet and the set of primitives provided by Myricom are not thread safe. However, we do believe that programmers could benefit from hybrid approaches because some applications can be easily structured as a set of concurrent/parallel tasks. That was the major motivation that led us to the investigation of a scalable communication strategy to support massive multithreaded applications in a cluster environment.

## 2 Background

Last decade many projects aimed to exploit the full computing power of networks of SMP workstations. In what follows we briefly present some key ideas that influenced nowadays cluster computing.

### 2.1 Distributed Multithreaded Programming

To run a distributed multithreaded program it is necessary to have a runtime system and a set of primitives to interface it[1]. Those primitives and their functionality highly influence the way programmers structure distributed applications.

MPI [21] programmers structure their applications according to the SPMD model and they are familiar with processor-to-processor message passing. PVM [10] permits some high level abstractions by introducing the notion of task. Communication takes place between tasks. The runtime system maps tasks to hosts.

Other platforms like TPVM [8], LPVM [23], a modified version of P4 [7], Chant [12] and Athapascan-0 [6] allow the creation of multiple threads. Communication occurs between threads using thread identifiers and send/receive primitives. Athapascan adds the concept of ports and requests: ports are independent

[1] Some distributed programming environments also include specific compilers.

from threads and so any thread can receive a message sent to a particular port; requests are used to test termination of asynchronous communication.

Panda [4], PM2 [16] and Nexus [9] also include thread support but they manage communication in a different manner; messages are delivered executing handlers previously registered by the user. This way programs are not forced to explicitly receive messages (via blocking or nonblocking primitives). These run-time systems are also able to automatically launch threads to execute handlers.

Remote service requests are another paradigm for remote execution and data exchange that some platforms do support. RPCs are asynchronous and match perfectly the communication paradigm of Panda, PM2 and Nexus, which obviously support this facility. Chant and Athapascan also implement RPCs.

Nexus provides an extra abstraction - the context - used to group a set of threads, which is an important concept for structuring applications. A context is mapped to a single node.

For thread support two different approaches may be used: developing a thread library or selecting an existent one. Panda and PM2 developed specific thread libraries in order to integrate communication and multithreading in a more efficient way. Chant manipulates the scheduler of existing thread packages (pthreads, cthreads, etc) to take message polling into account when scheduling ready threads. Cilk [5], which provides an abstraction to threads in explicit continuation-passing style, includes a work-stealing scheduler.

## 2.2 Efficient Message Handling

Using recent communication hardware, it is possible to send a message from one host to another in a few microseconds while throughput between hosts can achieve hundreds of Mbytes[2].

However, operating systems usually take advantage of internal buffers and complex scheduling techniques to deliver data to user level programs. For that reason low-level communication libraries have been developed to directly interface the hardware. GM [15], BIP [11] and LFC [2] are communication libraries that take full advantage from Myrinet technology, by means of zero-copy communication.

On the other hand, distributed applications manipulate complex entities and use several threads/processes of control. Messages incoming to a specific host must be redirected to the right end-point and so context-switching overheads may decrease performance. Active messages [22] are a well-known mechanism to eliminate extra overheads on message handling. Upcalls and popup threads are two techniques to execute message handlers [3] used in Panda.

The choice between polling or interrupts for message reception [13] may also have significant impact on program performance. LFC uses both mechanisms, switching from one to another according to the system status.

[2] Myrinet latency is less then $10\mu$s and one-way throughput is near 250MB/s.

### 2.3 pCoR Approach

pCoR aims to efficiently combine existent POSIX threads implementations (kernel Linux Threads, for example) and low-level communication facilities provided by hardware vendors (GM, for example). The goal is to provide a platform suitable for the development and execution of complex applications but we do not intend to directly support threads or to develop specific communication drivers.

Using Linux Threads we can take full advantage of multiprocessor systems and ensure compatibility with existent sequential routines. By implementing traditional send/receive primitives over a well-supported low-level communication facility as GM we guarantee performance and extendibility.

## 3 Thread-to-Thread Communication

pCoR runtime system distinguishes between inter and intra-node communication. Intra-node communication may occur between threads sharing the same address space (intra-process communication) or between threads from different processes (inter-process communication).

To manage communication, pCoR runtime system must be aware of thread location in order to select the most efficient mechanism for data sending. As a consequence the communication subsystem must be perfectly integrated on pCoR runtime system. It would be particularly difficult to use an existent thread-to-thread communication facility in an efficient manner because it would be necessary to integrate it with pCoR naming service.

At present we support two ports: UDP (for Ethernet devices) and GM (for Myrinet hardware).

### 3.1 Communication Channels

The development of a communication library to overcome pCoR communication needs must address two main issues:

1. identification – global process and thread identifiers, provided by pCoR resource manager, must be combined to produce unique identifiers to assign to communication end-points;
2. port virtualisation – low-level communication libraries to interface network adapters provide port abstractions to identify receivers and senders, but those abstractions are limited in number (GM library, for instance, only supports up to 8 ports).

In pCoR, identification is handled by a distributed service running on every process belonging to the same application. Basically, this is a directory service responsible to map pCoR identifiers into low-level identifiers used to route data at network interface level. To route information between components of the directory service, pCoR uses alternative communication facilities over TCP/IP. The impact of that solution is minimized through the use of local caches.

Port virtualisation will be explained in section 4.

### 3.2 Low-Level Interface

Communication between pCoR remote entities is implemented through a few primitives that use GM facilities to send and receive data. Although CoR specifies high-level abstractions to interconnect computing resources, it is possible to use these primitives to transmit and receive data in pCoR applications.

Senders must specify the destination using a pair `<pCoR process id, pCoR thread id>`, a tag and the address of the data to be sent. Data can be copied from its original address or it can be directly delivered to the GM library if it resides on a DMAble memory block. The reciprocal is valid for receivers.

Because both send and receive primitives are asynchronous, a test communication primitive with two modes – blocking or non-blocking – is provided.

```
int hndl   = sendCopy(int trgt_pid, int trgt_thid, int tag, void *data,
                  int size)
             sendDMA(...)
int hndl   = recvCopy(int src_pid, int src_pid, int tag, void *data,
                  int size, int *apid, int *athid, int *atag, int *asize)
             recvDMA(..., void **data, ...)
int status = testHandle(int handle, int mode)
```

## 4 Message Dispatching

Port virtualisation introduces the need to create a dispatching mechanism to handle messages from/to an arbitrary number of entities. Our approach uses a dispatcher thread per port to make possible several threads to share the same communication facility.

### 4.1 Dispatcher Thread

Send and receive primitives, executed by concurrent/parallel threads, interact with the dispatcher thread through queues. The send primitive enqueues messages for future dispatch whereas the receive primitive dequeues messages if any is available. Synchronous operation is supported through thread blocking mechanisms. Figure 1 shows the main aspects of message dispatching.

The dispatcher thread detects message arrival, via GM, using polling or blocking primitives. Every new message arriving to a port is enqueued in the receive queue and blocked threads (waiting for specific messages) are awakened. Whenever pending messages are detected in the send queue, the dispatcher thread initiates their transmission via GM.

Since we provide two approaches[3] to interface GM – polling and blocking primitives – the dispatcher operates in one of two modes: non-blocking or blocking.

[3] Currently available as compile options.

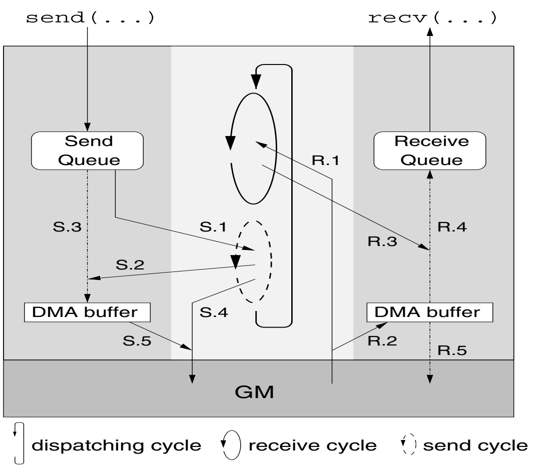


**Fig. 1.** Message dispatching mechanism

The non-blocking dispatcher uses a sole thread to execute an infinite loop sending and receiving messages. After polling the GM port for events[4] the dispatcher tries to find messages to transmit.

The blocking dispatcher must overcome a basic problem: if pCoR blocks itself waiting for message arrival, it will be impossible to send out any messages until a network event occurs because GM primitives are not thread safe. Experience proved that if a thread is blocked (after calling a GM blocking primitive) it is possible for another thread to send out messages if we use an additional GM port. Thus the blocking dispatcher uses two threads and two ports – one to receive and another to send messages. A thread waits for messages (from other nodes) issuing a GM blocking primitive while the other blocks itself waiting for messages to be sent to other nodes.

## 4.2 Segmentation and Retransmission

To transmit messages over GM, it is necessary to copy data into DMAble memory blocks[5]. pCoR supports the transmission of arbitrary size messages, i.e., the communication layer must allocate DMAble arbitrary size buffers. Because processes cannot register all their memory as DMAble, we use buffers up to 64kbytes requested on library start-up. This means that long messages must be segmented.

Segmentation involves reassembling message fragments at destination and it implies that sequence numbering to identify fragments belonging to the same message is needed. Sequence numbers are managed by the interface developed to

[4] GM events signal network activity (message arrival, acknowledgment, etc).
[5] Program data stored in DMAble memory is transmitted as a zero copy message.

manage the queues used by the dispatcher. Every fragment is handled as a simple message by the dispatcher; dequeue and enqueue operations are responsible for fragmentation and reassembling.

Message sequencing is used to overcome another problem: fragment/message retransmission. Although GM guarantees the correct delivery of messages, the lack of resources at destination may not permit reception at a specific time. In those cases it is necessary to retry transmission after a certain period of time.

### 4.3 Multiple Routes and Message Forwarding

Cluster nodes may have installed multiple network interfaces from different vendors[6]. It is also possible that not all nodes from a cluster share a common communication technology. Even clusters on different locations may be interconnected using Internet protocols.

For those scenarios, it is desirable to allow computing entities to select at runtime the appropriate communication protocol and to provide forwarding capabilities to overcome cluster partitions (Madeleine [20] addresses these topics). It is also important to provide mechanisms to choose the better location for computing threads according to host-to-host link capabilities. For instance, for a cluster fully connected with Fast Ethernet but having part of the nodes connected with Myrinet, it would be desirable to have the runtime system responsible to start on Myrinet nodes those threads with higher communication demands.

pCoR uses a straightforward mechanism to provide multiple routes on heterogeneous clusters. At start-up each node registers its communication ports and builds a simple routing table containing information about protocols and gateways available to reach each remote node. As pCoR allows to dynamically add nodes to an application, the runtime system rebuilds the routing table at each node every time a start-up event is received.

Message forwarding is accomplished by the dispatcher thread. pCoR message headers include the final destination (process id) of the message along with the information pointed out in figure 2.

## 5 Data Structures

Message dispatching requires appropriate data structures to guarantee low-latency reception and sending. The pCoR communication layer architecture uses two main queues per port to store messages. Those queues must minimize required memory size and must permit fast access to store/retrieve data.

### 5.1 Message Reception

The `recv` primitive used in pCoR, executed concurrently by an arbitrary number of threads, searches for messages according to certain attributes: originator

[6] It's common to connect cluster nodes to both Ethernet and Myrinet switches.

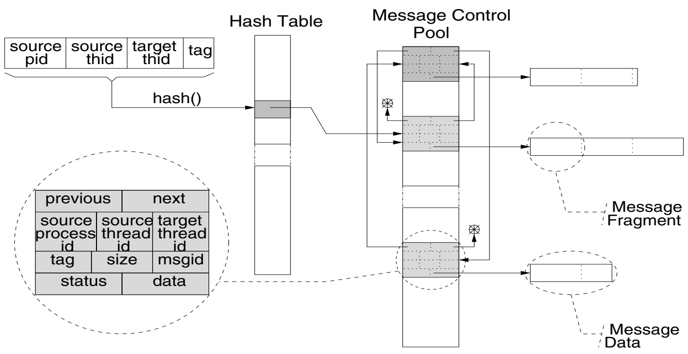


**Fig. 2.** Data structures for message reception

process, originator thread and message tag. As we use an only receive queue per process the destination thread identifier is also automatically included to search for a specific message.

A tuple `<source process, src. thread, target thread, tag>` is used to calculate a hash index to access a vector of pointers to message control blocks. The message control blocks are stored in a fixed size array which means that a limited number of messages can be pending for reception. Collisions resulting from the application of the hash function and messages addressed to the same thread from the same origin and with the same tag are managed as a linked list of message control blocks as shown in figure 2.

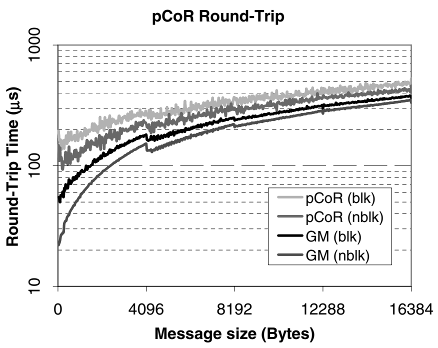
Message control blocks contain message attributes, a pointer to the message data, sequencing information and fragmentation status. For fragment tracking 32 bits are used – 1 bit for each fragment – supporting messages up to 2095872 bytes[7].

## 5.2 Message Sending

The `send` primitive enqueues messages for future dispatch whereas the dispatcher thread dequeues those messages for sending over GM. Because message dispatching uses FIFO order, at first sight we might think that a simple queue would be adequate to hold pending messages. However, since segmentation and retransmission are provided, the dispatcher needs some mechanism to access a specific message. Actually, segmentation requires the ability to dequeue single message fragments whereas delivery acknowledgment events from GM layer, handled by the dispatcher, require the ability to set message status for a specific message.

For short, data structures for message sending will be analogous to those used for message reception, but it is necessary to have a dequeue operation performing according to FIFO.

[7] Maximum message size results from $(64k - (fragmentheadersize)) * 32$.



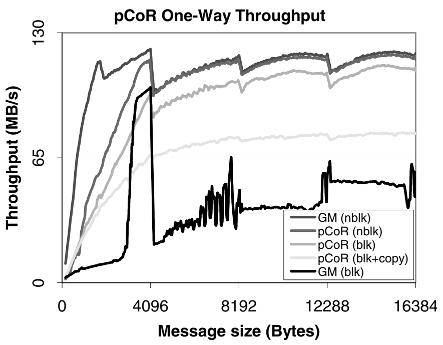


**Fig. 3.** Round-Trip and Throughput in pCoR

## 6 pCoR Raw Performance

Although pCoR provides high-level abstractions, like remote thread creation and definition of complex organizers, it is important to evaluate the raw performance obtained when transferring data between threads.

The results we present were obtained using a simple benchmark program that engages on communication two pCoR tasks (threads) executing on different machines. We used two dual PIII 733MHz workstations, connected by Myrinet (LANai9 64bits/66MHz interfaces), running Linux RedHat 7.2 (kernel 2.4.7-10smp). The tests were performed with no additional load at each workstation.

Figure 3 presents round-trip times and one-way throughput for messages from 1byte to 16kbytes. Values for the GM low-level interface performance (host-to-host communication) are also presented to better understand the overhead of thread-to-thread communication. The experiments took into account the two mechanisms GM provides to receive events - polling and blocking receives[8].

It is important to note the impact of message dispatching[9]. For each message exchange, the pCoR runtime system must wake up two blocked threads; the dispatcher must signal the message arrival to a specific thread. Using some simple POSIX threads tests, we evaluated the overhead of waking up a thread blocked on a condition variable (using linuxthreads-0.9, libc-2.2.4). We concluded that this overhead exceeds 35$\mu$s. This explains round-trip times obtained in pCoR; a message exchange in pCoR incurs in a 70$\mu$s penalty due to thread wake up.

It is also important to note the result of using blocking primitives to interface the GM library. Although the use of blocking primitives has the advantage of freeing the processor for useful computation, message reception incurs in a 15$\mu$s penalty (30$\mu$s for a message exchange) due to interrupt dispatching.

[8] In the charts legends *blk* and *nblk* stands for blocking and non-blocking.

[9] Legend items order correspond to the placement of chart curves; the top curve corresponds to the first legend item and vice-versa.

Throughput tests showed that GM guarantees 120Mbytes/s[10] using non-blocking primitives (polling). The use of GM blocking primitives produces poor and unpredictable results. pCoR can achieve almost the same throughput as GM for messages longer than 4kbytes and the use of blocking primitives did not produce the same negative impact that we noticed when using GM directly.

Surprising results were obtained when we decided to test the pCoR non-zero-copy communication primitives[11]. For data residing on non-DMAble memory, pCoR must allocate a memory block and perform a data copy. In spite of this overhead, pCoR outperforms the throughput obtained in GM host-to-host tests using blocking primitives.

We conclude that GM blocking primitives can behave nicely when several threads share the same processor.

## 7 Case Study

To emphasize the importance of thread-to-thread communication we present an application intended to manage (display) huge maps. Those maps are composed of several 640x480 pixel JPEG images.

In our case study we used a 9600x9600 pixel map consisting of a 15x20 matrix of JPEG images. The main objective is the visualization of arbitrarily large map regions. Regions larger than the window size require the images to be scaled down.

The architecture we propose to manage this kind of maps takes into account the following requisites: high computing power to scale down images, large hard disk capacity to store images and high I/O bandwidth to load JPEG images from disk.

### 7.1 Multithreading

Assuming we have an SMP machine with enough resources to handle those huge maps a multithreaded solution can be developed to accelerate the decompression of JPEG images and the reduction of image bitmaps.

Figure 4 shows two C++ classes used to model a simple multithreaded solution. An object `imgViewer` is used to display a map region, according to a specified window size, after creating the corresponding bitmap. The bitmap is created using an object `imgLoader` which creates a thread to execute the method `startFragLoad`. The `imgViewer` calls the method `startFragLoad` from class `imgLoader` for each JPEG image required to create the final bitmap.

To display a 9600x9600 pixel map region, for instance, 300 threads will be created to load the corresponding JPEG images and to scale them down. Using a 600x600 window to display the final bitmap, each thread will scale down 16

[10] Our Myrinet configuration would reach 1.28Gbits/s, due to switch constraints, but the workstations PCI bus cannot guarantee such performance.

[11] In the legend of the throughput graph *blk+copy* stands for blocking with buffer copy.

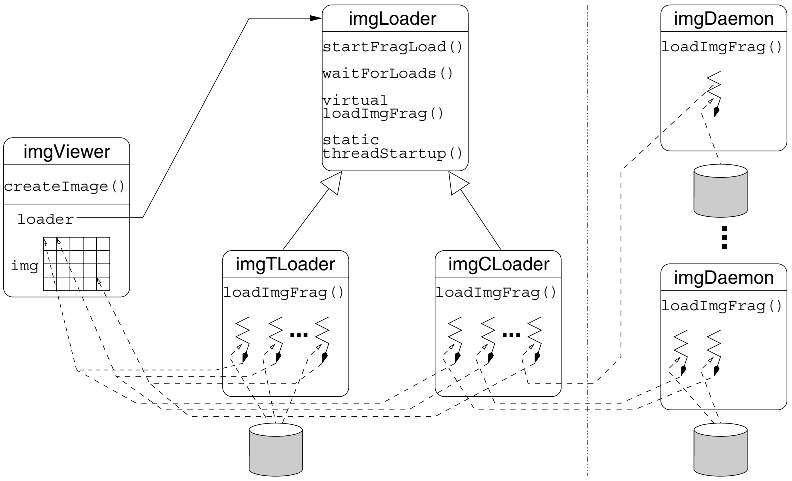


**Fig. 4.** Object model for multithread loading of huge maps

times an original 640x480 JPEG image in order to produce a 40x30 bitmap fragment. The object `imgViewer` is responsible for bitmap fragment reassembling.

## 7.2 Scalable Multithreading

Assuming we have a cluster with enough disk space at each computing node it is possible to spread all the JPEG images across all the nodes. Thus we will overcome disk capacity limitations and each node will be able to produce local results, without requesting images from a centralized server, taking advantage from cluster nodes computing power and local I/O bandwidth. Of course we will need some mechanism to discover which node holds a specific image, but it can be done using a simple hash function.

Figure 4 depicts `imgDaemon` object instances corresponding to daemons running on each cluster node to load and transform images according to requests received from a remote `imgLoader`. The `imgLoader` used in our cluster environment requests bitmap fragments from remote cluster nodes instead of loading it directly from disk.

The `imgLoader` class is in fact a virtual class used to derive two classes:

1. `imgTLoader` – multithreaded loader to use in a single SMP machine;
2. `imgCLoader` – multithreaded broker to use in a cluster.

Note that the development of the multithreaded solution to use in a cluster environment, assuming we had already developed a solution to use in a single SMP machine, was a trivial task:

– a virtual class `imgLoader` was introduced to permit the use of the same `imgViewer` object;

**Table 1.** Hardware specifications

| Specifications | SMP server | cluster node |
|---|---|---|
| Processor | 4x Xeon 700MHz | 2x PIII 733MHz |
| Memory | 1Gbyte | 512Mbytes |
| Hard Disk | Ultra SCSI 160 | UDMA 66 |
| Network | LANai9 Myrinet, 64bits/66MHz | |

– a new class `imgCLoader` was derived to handle requests and to receive data from remote threads;
– the code from class `imgTLoader` responsible for JPEG image loading and scaling down is placed in a daemon program (object `imgDaemon`) to execute at each cluster node.

This approach can be used to scale many multithreaded applications primarily developed to use in a single SMP machine.

## 7.3 Performance Evaluation

Performance evaluation was undertaken using a four node Myrinet cluster and an SMP server connected to cluster nodes, all running Linux. Table 1 summarises hardware specifications for our test bed.

Figure 5 presents computation times required to display 7 different map regions using a 600x600 window. The left side of the figure shows 7 map regions consisting of 2 to 300 JPEG images. Those regions, marked from 1 to 7, correspond respectively to 1:1, 1:2, 1:4, 1:5, 1:8, 1:10 and 1:16 scaling factors. The right side of the figure presents the results obtained using:

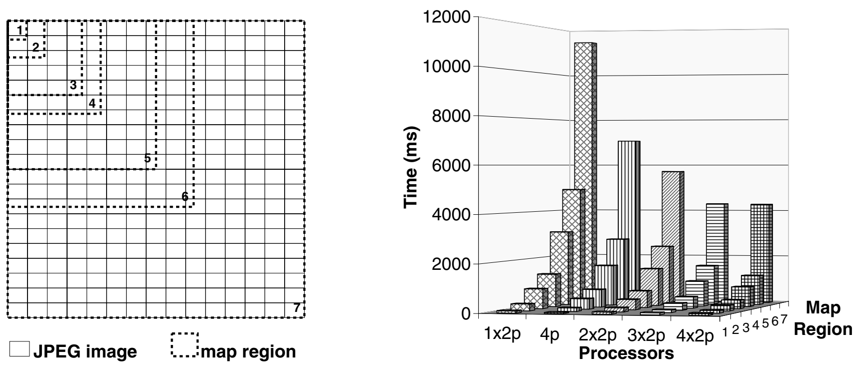


**Fig. 5.** Performance measurements for differrent scenarios

– a single 4-processor SMP machine (an `imgTLoader` object instance is used by the application) identified as `4p`;
– 1 to 4 cluster nodes (an `imgCLoader` object instance is used by the application) identified as `1x2p`, `2x2p`, `3x2p` and `4x2p`.

It is important to point out that the results obtained using the cluster solution based on 2 nodes (4 processors) supersede the results from the multithreaded solution based on a 4-processor SMP server. The main cause is the higher bandwidth available to load JPEG images from disk.

It is also important to emphasize the results obtained using the cluster solution based on 4 nodes (8 processors). As expected better performance was achieved for the majority of region maps tested, but it was not possible to outperform the result achieved with 3 cluster nodes for 9600x9600 region maps. That happens because the object `imgCLoader`, executing 300 threads to receive results from cluster nodes, is not fast enough to process incoming messages because of thread contention accessing communication library.

## 8 Conclusions

Using the current pCoR implementation it is possible to achieve communication between threads residing on any node of a cluster.

Thread scheduling is still a high CPU consuming task, particularly when using Linux Threads. Port virtualisation is consequently somewhat inefficient. Nevertheless, we do believe that it is convenient to program multithreading solutions to run in a cluster environment using Linux kernel threads because they can take full advantage of multiprocessor systems and I/O can easily overlap computation.

For applications demanding a high level of parallelism it is possible to develop traditional multithreaded solutions (to use in a single SMP machine). Considering that in most cases data sharing among threads is not a high requisite, because data can be easily spread among computational entities, it is possible to implement thread synchronization using messages. For those applications pCoR provides support for scalable multithreading.

## References

[1] A. Alves, A. Pina, V. Oliveira, and C. Moreira. CoR's Faster Route over Myrinet. In *MUG '00 - First Myrinet User Group Conference*, pages 173–179, 2000. 580

[2] R. Bhoedjang. *Communication Architectures for Parallel-Programming Systems.* PhD thesis, Advanced School for Computing and Imaging, Vrije Universiteit, 2000. 581

[3] R. Bhoedjang and K. Langendoen. Friendly and Efficient Message Handling. In *29th Hawaii International Conference on System Science*, pages 121–130, 1996. 581

[4] R. Bhoedjang, T. Rühl, R. Hofman, K. Langendoen, and H. Bal. Panda: A Portable Platform to Support Parallel Programming Languages. In *USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, 1993. 581

[5] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996. 581

[6] J. Briat, I. Ginzburg, and M. Pasin. *Athapascan-0 User Manual*, 1998. 580

[7] A. Chowdappa, A. Skjellum, and N. Doss. Thread-safe message passing with p4 and MPI. Technical report, Computer Science Department and NSF Engineering Research Center, Mississippi State University, 1994. 580

[8] J. Ferrari and V. Sunderam. TPVM: Distributed Concurrent Computing with Lightweight Processes. In *4th IEEE Int. Symposium on High Performance Dist. Computing - HPDC '95*, 1995. 580

[9] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, 1996. 581

[10] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.* Scientific and Engineering Computation. MIT Pres, 1994. 580

[11] P. Geoffray, L. Prylli, and B. Tourancheau. BIP-SMP: High Performance Message Passing over a Cluster of Commodity SMPs. In *SC99: High Performance Networking and Computing Conference*, 1999. 581

[12] M. Haines, D. Cronk, and P. Mehrotra. On the Design of Chant: A Talking Threads Package. In *Supercomputing '94*, 1994. 580

[13] K. Langendoen, J. Romein, R. Bhoedjang, and H. Bal. Integrating Polling, Interrupts, and Thread Management. In *6th Symp. on the Frontiers of Massively Parallel Computing*, 1996. 581

[14] C. Moreira. CoRes - Computação Orientada ao Recurso - uma Especificação. Master's thesis, Universidade do Minho, 2001. 579

[15] Myricom. *The GM Message Passing System*, 2000. 581

[16] R. Namyst and J. Méhaut. PM$^2$: Parallel Multithreaded Machine. A computing environment for distributed architectures. In *ParCo'95*, 1995. 581

[17] A. Pina. *MC$^2$ - Modelo de Computação Celular. Origem e Evolução.* PhD thesis, Departamento de Informática, Universidade do Minho, Braga, Portugal, 1997. 579

[18] A. Pina, V. Oliveira, C. Moreira, and A. Alves. pCoR - a Prototype for Resource Oriented Computing. In *Seventh International Conference on Applications of High-Performance Computers in Engineering*, 2002. 579

[19] A. Pina, J. Rufino, A. Alves, and J. Exposto. Distributed Hash-Tables. PADDA Workshop, Munich, 2001. 580

[20] B. Planquelle, J. Méhaut, and N. Revol. Multi-protocol communications and high speed networks. In *Euro-Par '99*, 1999. 585

[21] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI - The Complete Reference.* Scientific and Engineering Computation. MIT Pres, 1998. 580

[22] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992. 581

[23] H. Zhou and A. Geist. LPVM: A Step Towards Multithread PVM. *Concurrency: Practice and Experience*, 10(5):407–416, 1998. 580