RoCL: A Resource Oriented Communication Library

Albano Alves[1], António Pina[2], José Exposto, and José Rufino

[1] Instituto Politécnico de Bragança,
Campus Sta. Apolónia, 5301-857 Bragança-Portugal
albano@ipb.pt
[2] Universidade do Minho
pina@di.uminho.pt

Abstract. RoCL is a communication library that aims to exploit the low-level communication facilities of today's cluster networking hardware and to merge, via the resource oriented paradigm, those facilities and the high-level degree of parallelism achieved on SMP systems through multi-threading.
The communication model defines three major entities – contexts, resources and buffers – which permit the design of high-level solutions. A low-level distributed directory is used to support resource registering and discovering.
The usefulness and applicability of RoCL is briefly addressed through a basic modelling example – the implementation of TPVM over RoCL. Performance results for Myrinet and Gigabit Ethernet, currently supported in RoCL through GM and MVIA, respectively, are also presented.

Keywords: cluster computing, message-passing, directory, multi-threading.

1 Introduction

The appearing of commodity SMP workstations and high-performance SANs aroused the interest of researchers to the topic of cluster computing. Important tools have been developed and have provided an inexpensive vehicle for some classical problems. However, to address an important class of non-scientific large-scale multi-threaded applications and achieve the desired efficiency, in the presence of multiple communication technologies, current approaches and paradigms are not adequate.

1.1 Inter-node Communication

Cluster nodes are typically interconnected by means of high-performance networks like Myrinet or Gigabit Ethernet, but it is also very common to have an alternate low cost communication facility, like Fast Ethernet, to handle node setup and management. To interface high-performance NICs, user level communication libraries, like GM [9] and MVIA [11] (a VIA [4] implementation), became the right choice because it is possible to avoid context switching and memory copies.

Several runtime systems and programming environments have been ported to exploit those technologies (MPI over GM, MPI over Infiniband, PVM over VIA, etc). To provide a uniform interface to multiple communication technologies, intermediate-level communication libraries, such as Madeleine [3], had also been developed. Nevertheless,

H. Kosch, L. Böszörményi, H. Hellwagner (Eds.): Euro-Par 2003, LNCS 2790, pp. 969–979, 2003.

it is not usual to combine multiple technologies in order to speed up application execution. Moreover, exploiting a secondary network like Fast Ethernet to perform specific communication tasks to alleviate the overhead of the main networking hardware seems to be an interesting topic that no one has properly addressed yet.

The point is, low-level communication libraries allow the exploitation of high-performance networking hardware but their interfaces are inadequate for application programming and their communication models, mainly concerned with node-to-node message exchanging, make difficult the integration with higher-level abstractions. For instance, GM supports up to eight communication end-points per node which are not enough to deal with the requirements of highly multi-threaded applications.

1.2 Execution Environment

Traditionally the use of parallelism and high-performance computation has been directed to the development of scientific and engineering applications. Multiple platforms are available to help programmers design and develop their applications but it is important to note that the majority of the runtime environments are practically static, since application modules cannot be started disregarding the running ones; in a PVM application, for example, an initial task launches some other tasks (processes), which in turn may launch other ones, and the identifiers of relevant tasks have to be announced by their creators to other participants; in a PM2 [10] application, for example, the programmer is responsible for defining the routines that will be used to deliver active messages.

To deal with the growing complexity of today's large-scale parallel applications, as is the case of a system that integrates crawling, indexing and querying facilities, it is necessary to have a flexible execution environment where: multiple applications from multiple users collaborate to reach a common goal; I/O operations (access to databases, for example) represent a significant part of the application execution time; system modules may execute intermittently; application requirements may vary unpredictably.

Execution environments that fulfil these requirements may be developed using the high-level abstractions provided by MPI, PANDA or others. However, the use of an existing platform imposes some constraints that are incompatible with the innovation needed to deal with the requirements of novel and more complex applications.

2 Resource Oriented Communication

RoCL, the Resource oriented Communication Library we present in this paper, uses existing low-level communication libraries to interface networking hardware. As a new intermediate-level communication library it offers a novel approach to system programmers to facilitate the development of a higher-level programming environment that supports the resource oriented computation paradigm of CoR[12].

2.1 General Concepts

RoCL communication model defines three major entities – contexts, resources and buffers – and uses the services provided by a specific low-level distributed directory.

A context is defined whenever the library is started up by calling the appropriate *init* primitive. Every context owns one or more low-level communication ports to send/receive messages, acting as a message store.

Resources are a common metaphor used to model both communication and computation entities whose existence is announced by registering them in a global distributed directory service. Every resource is associated to an existing context and possesses a unique identifier. RoCL does not define the properties of resources neither limits their definition. Resources are instances of application level concepts whose idiosyncrasies result from a set of attributes specified at creation. An attribute is a pair $\langle name, value \rangle$ where name is a string and value is a byte sequence. A resource R with n attributes is defined by the expression $R = \{\langle name_1, value_1 \rangle, ..., \langle name_n, value_n \rangle\}$. To register a specific resource, the programmer must enumerate its attributes (see 3.1).

To minimize memory allocation and registering operations, RoCL uses a buffer management system. Messages are held on specific registered memory areas to allow zero-copy communication. Prior to sending messages the programmer must acquire adequate size buffers. At reception the library is responsible for providing the communication subsystems with the necessary pre-registered memory blocks.

Resource global identifiers are used to determine the origin and destination of messages. The identity of a resource may be previously known by the application or it may be obtained from the directory by querying it.

Figure 1 presents the steps required for the operation of a basic client/server interaction, according to the RoCL communication model.

Server

1. initialize a RoCL context
2. register resource $R_1 = \{\langle type, server \rangle\}$ and store the returned identifier in gid_1
3. wait for any message sent to gid_1 and store the buffer pointer in msg_1
4. process message data through msg_1
5. return buffer msg_1 to the library

Client

1. initialize a RoCL context
2. register resource $R_2 = \{\langle type, client \rangle\}$ and store the returned identifier in gid_2
3. discover resource $\{\langle type, server \rangle\}$ and store the returned identifier in gid_3
4. obtain a buffer with a specific size and store the returned pointer in msg_2
5. prepare message data by using msg_2
6. send msg_2 from gid_2 to gid_3
7. return buffer msg_2 to the library

Fig. 1. A basic modelling example.

2.2 Basic Interface

The basic set of primitives that programmers may use to exploit RoCL is presented in table 1, organized according to the involved RoCL entities. Resource handling primitives are discussed through section 3. Buffer management and communication primitives have been presented in detail in a previous paper (see [1]).

In this context it is important to highlight that buffer and communication handling primitives were designed to keep up zero-copy messaging. The use of low-level communication libraries like GM and VIA does not automatically guarantee zero-copy communication; the higher-level abstraction layer must define an appropriate interface to preserve the low-level features. SOVIA [8] and PVM over VIA [5], for example, use VIA, which permits zero-copy communication, but because users may continue to use the traditional sockets and PVM interfaces, those systems must copy or register user data (memory regions) before sending and can not avoid one copy at reception.

Table 1. RoCL basic primitives.

Contexts
`int rocl_init()`
`rocl_exit()`
Resources
`int rocl_register(rocl_attrl_t *attrs)`
`rocl_delete(int gid)`
`int rocl_query(int *gid, rocl_attrl_t *attrs)`
Buffers
`void * rocl_bfget(int len)`
`rocl_bfret(void *ptr)`
`rocl_bftoret(void *ptr)`
`int rocl_bfstat(void *ptr)`
Communication
`rocl_send(int ogid, int dgid, int tag, void *ptr, int len)`

`int rocl_recv(int dgid, int ogid, int tag, void **ptr,`
`int *aogid, int *atag, int *alen, int timeout)`

3 Directory Service

RoCL creates a fully dynamic system where communication entities may appear and disappear, at any moment, during application execution. To support these features we use the resource abstraction along with the facilities of a global distributed directory service. The importance of such a service is emphasized in [2] and [7].

RoCL directory service is a global distributed system that provides efficient and scalable access to the information about registered resources. This service enables the development of more flexible distributed computing services and applications.

3.1 Attribute Lists

A resource is defined/registered by specifying an attribute list. The primitives used to manipulate resource attribute lists are presented in table 2.

Attribute lists are used both for resource registering and querying. To successfully register a resource, all its attributes have to be completely specified, i.e. each attribute has to have a name and a value. For querying purposes, some attributes may be partially defined, i.e. attribute values may be omitted (*NULL* values) in order to inform the library about the attributes we want to know for a specific resource.

Table 2. RoCL primitives to handle attribute lists.

```
rocl_attrl_t * rocl_new_attrl(int max_len)
int rocl_add_attr(rocl_attrl_t *attrs, char *name, void *val,
                          int len)
void * rocl_get_attr(rocl_attrl_t *attrs, char *name, int *len)
rocl_kill_attrl(rocl_attrl_t *attrs)
```

An attribute list is stored in a contiguous memory region to avoid memory copies when sending it to a server (see 3.2). In fact, an attribute list itself is used as a request/reply packet, requiring some space to be reserved at the attribute list head to allow the attachment of control information.

3.2 Local Operation

RoCL resources are registered at each cluster node using a local server that obtains a subset of global identifiers at start-up, thus minimizing inter-node communication (figure 2). The local resource database (DB in figure 2) is maintained in main memory and hashing techniques are used to accelerate query operations.

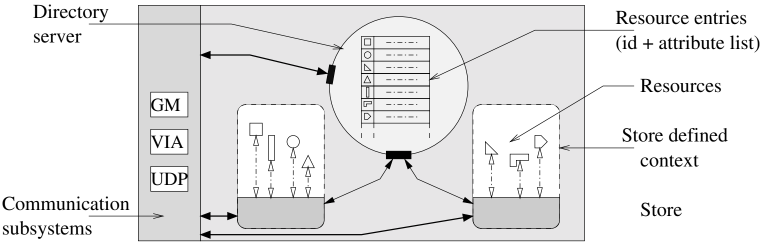

Fig. 2. Resource local registering.

A query received by a local server corresponds to a request packet that may contain: the resource identifier, some completely specified attributes (with valid names and values) and some partially specified attributes (with *NULL* values). If the resource identifier is present, the search mechanism is trivial; otherwise the completely specified attributes are used to produce hash indexes. After finding the right resource, all partially specified attributes are examined and each one will be completed if an attribute with a matching name was previously registered for that resource. Because the library reserves space in the attribute list to store expected values for incomplete attributes, request packets may be used as reply packets avoiding memory allocation and copying.

3.3 Global Operation

If a particular query can not be satisfied at the local server, a global search is initiated. In a global search, all servers running across the cluster receive the request but only the one where the query succeeds is committed to reply.

As a first approach to support global searches, we used UDP broadcast to spread requests through the Fast Ethernet network. This approach benefits from the native broadcasting support at protocol and hardware level.

Requests may also be delivered to servers by combining UDP broadcast and GM or VIA spanning trees. The general operation will be as follows: 1) local servers periodically announce their presence using UDP broadcast; 2) each server maintains a list with all active servers; 3) spanning trees are used to reach all active servers. The use of spanning trees results from the fact that Myrinet hardware does not support broadcasting and the connection oriented model, adopted by VIA, is not compatible with multicasting.

3.4 Multiple-Answer Queries

Queries that don't specify a resource identifier may result in multiple answers returned by one or more servers. This happens because different resources may share some (or even all) attribute names and values.

RoCL provides dedicated primitives, (see table 3) to manage multiple answers to a single query. This interface should not be used whenever we just want a single answer or if it is known in advance that there will be a sole answer to a particular query.

Table 3. RoCL primitives to handle multiple-answer queries.

```
rocl_handler_t * rocl_query_start(rocl_attrl_t *attrs)
int rocl_query_next(rocl_handler_t *handler, rocl_attrl_t *attrs)
int rocl_query_stop(rocl_handler_t *handler)
```

When using this interface, results are fetched from the local server, one at a time, as if multiple independent single answer queries were in progress. Each request/reply packet transports the data – an attribute list – corresponding to a single resource.

To support multiple answers, the local server maintains some control information for each query that is in progress, in order to be able to decide to: look for an answer in the local database, broadcast the query, store the answers returned by remote servers, search the local database for the next result, return an answer obtained from a remote server or request the next result from a remote server.

Each remote server may return only one result as a reply to a specific broadcast. The local server stores the multiple answers received and sends a unicast request to a specific remote server whenever a result previously returned by that server is used up.

4 Inter-resource Message Passing

RoCL applications address messages to resources previously located using the directory service. Inter-resource message passing raises two main problems: message routing, because there is not a direct mapping between resources and communication subsystem addresses, and message dispatching, because resources are animated by threads and multiple low-level communication ports must be multiplexed.

4.1 Message Routing

Contexts are the only valid end-points known by the communication subsystems. Therefore, resources must be mapped into contexts before message sending.

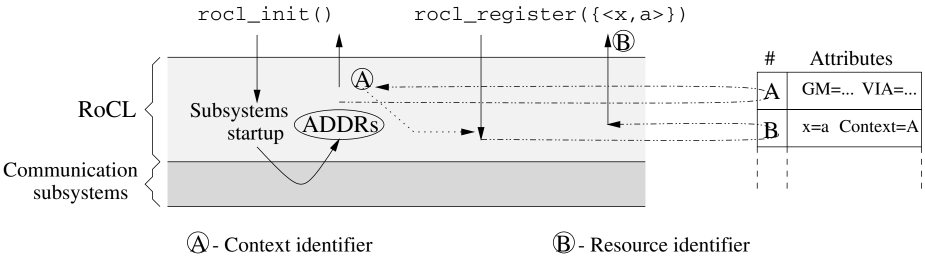

Fig. 3. Resource-context mapping.

The mapping between resources and contexts is handled as shown in figure 3. Contexts are registered at library initialization; they are managed as system resources. The addresses (or ports) of the communication subsystems are used as context attributes. The RoCL library uses the context global identifier, returned by the directory service, as an automatic attribute for resources, i.e. all resources will be tagged with the identifier assigned to the context where they belong.

This approach is quite inefficient because three steps will be required to send a message to a particular resource: 1) the resource context must be obtained by querying the directory system, 2) the context addresses must also be obtained and finally 3) the message will be sent. The two first steps will require some messages (requests and replies) to be exchanged and therefore communication latency will be unacceptable. To overcome this problem, the library uses two dedicated caches to store the most recently required mappings between resource identifiers and context identifiers and between context identifiers and their communication subsystem addresses.

4.2 Message Dispatching

RoCL resources are animated by threads, meaning that RoCL must support concurrent/parallel access to communication facilities. Besides that, message reception is totally asynchronous, meaning that message delivering must take into account that receivers may not be waiting the messages sent to them.

RoCL is a fully connectionless communication system that uses a dispatching mechanism based on system threads and message queues. System threads, one per communication subsystem, wait for messages using polling and interrupt handling techniques and store the received messages in a receiving queue. Resources access this queue to retrieve messages according to some selection criteria (message tag, origin identifier, etc). RoCL provides blocking and timed receivings, through the `timeout` parameter (see table 1): a negative timeout indicates a blocking behavior.

Sending primitives may directly access the communication subsystems but, whenever concurrent calls occur, messages are stored in a sending queue and the primitives return immediately.

Because receiving and sending queues only handle message descriptors, containing a pointer to the message data, no extra copies are introduced. A detailed explanation of message dispatching and the way it is related to quite distinct user-level communication protocols – GM and VIA – may be found in [1].

5 Applicability and Performance

Although RoCL was designed as an intermediate-level message-passing library to the development of a new programming platform that is still under construction, it constitutes per se a basic programming tool. So, it is already possible to present particular applicability examples along with raw performance results.

5.1 TPVM over RoCL

Assuming a simplified view of TPVM [6] we will examine the design/implementation of some of its basic functionality using RoCL primitives.

PVM tasks (processes under UNIX) will create a RoCL context at start-up when they start running. They also register themselves as task resources, using an attribute $\langle type, task\rangle$, and their global resource identifiers are used as PVM task identifiers (TIDs). A simplified PVM task spawning mechanism may be achieved as follows:

- the parent task uses *rsh* to execute a special process launcher passing to it the conventional arguments required to start the spawned program along with its TID;
- the launcher registers a temporary resource using as attributes its process identifier (PID) and the received parent task identifier ($\{\langle type, tmp\rangle, \langle pid, ...\rangle, \langle ptid, ...\rangle\}$);
- the launcher starts the target program using the *exec* primitive;
- the spawned task, at start-up, queries the directory service, using its process identifier, to find the temporary resource and to obtain the parent task identifier;
- the spawned task sends its global resource identifier to the parent/spawner process.

To remotely activate TPVM threads we will need a launcher thread, automatically created when the PVM process/task starts, that will block waiting for messages that request the activation of a specific TPVM thread. The launcher thread – the *pod* controller – registers itself as a system thread resource using an attribute $\langle type, systhread\rangle$.

The `tpvm_export` primitive will correspond to a simple RoCL register operation. A TPVM thread will be defined by using a name and the global resource identifier of its *pod* controller – $\{\langle type, threaddef\rangle, \langle name, ...\rangle, \langle systhread, ...\rangle\}$.

The `tpvm_spawn` primitive will use the thread name to query the directory service and find the global identifier of the *pod* controller. This global identifier is used to send a request message to the launcher thread, which will create the desired thread. Instantiated threads also register themselves using an attribute $\langle type, thread\rangle$ and send their global identifiers to the spawners. A spawned thread obtains its parent identifier directly; the request sent to the launcher thread carries the spawner global identifier.

A host database may also be provided as a collection of RoCL resources. However, since no PVM daemons are used, this database will only be useful to find out available machines and select a particular target node for spawning operations.

5.2 Inter-resource Message-Passing Performance

RoCL message-passing performance is influenced by the communication subsystems we support, the way we interface them and the capabilities of LINUX threads [1].

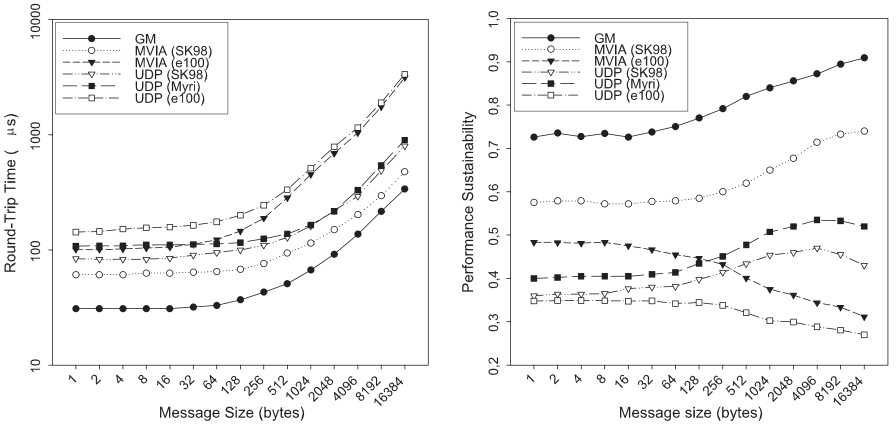

Fig. 4. RoCL performance.

Figure 4:*left* presents round-trip times relating to three networking technologies – Myrinet (LANai 9), Gigabit Ethernet (SysKonnect 9821) and Fast Ethernet (Intel EtherExpress PRO/100) – exploited through GM, MVIA and UDP [2]. Performance tests were performed on dual Pentium III 733MHz workstations running RedHat 9.0. Myrinet and Gigabit adapters were attached to 64bits/66MHz PCI slots.

RoCL over GM allows to achieve a 30μs round-trip time for 1 byte messages, which corresponds to an overhead of 10μs when compared to GM node-to-node performance. MVIA for SysKonnect, as expected, outperforms all communication subsystem alternatives but GM. Surprisingly, for small messages, SysKonnect hardware produces better results than Myrinet, when using UDP.

In order to evaluate message-passing alternatives, it is mandatory to also analyze the impact of communication on computation and vice-versa. To evaluate this interdependence we had calculated the execution rate of a particular computation cycle, using one thread per processor, without performing any communication task, and then we run both the original round-trip and the computation benchmarks concurrently.

The impact on computation may be expressed by the ratio Ex_{CS}/Ex_0, where Ex_{CS} stands for the execution rate obtained when we run, concurrently, the round-trip benchmark using one of the communication subsystem alternatives and Ex_0 stands

[1] Currently RoCL is only supported under LINUX.

[2] GM runs on Myrinet, MVIA runs on SysKonnect and Intel and UDP runs on each of them.

for the execution rate obtained without background communication. Similarly, the ratio Rt_{CS_0}/Rt_{CS}, where Rt_{CS} and Rt_{CS_0} stand for the round-trip times obtained for a given communication subsystem, respectively, with or without computations taking place concurrently, express the impact on communication. To easily compare round-trip ratios from different communication subsystems, we use the ratio Rt_{GM_0}/Rt_{CS}, where Rt_{GM_0} stands for the round-trip times obtained using GM (the best round-trip times we may achieve with RoCL).

These two ratios express the performance sustainability of computation and communication, when they use the same CPU(s). As we consider that both communication and computation performances are equally fundamental to determine the success of high performance computing, we use an overall ratio, for each communication subsystem, calculated as a geometric average.

Figure 4:*right* presents the overall performance sustainability we can expect to achieve in RoCL applications. It is important to note that the RoCL impact on computation is in accordance to the selected hardware and communication subsystem: FastEhternet adapters perform badly and require higher CPU intervention as message size increases; UDP, a complex protocol, requires more CPU cycles than MVIA and GM and so the overall performance drops.

6 Conclusions

RoCL introduces a new communication paradigm to facilitate the design and implementation of high-level execution environments. In this paper, the key concepts related to inter-resource message-passing and the operation of a low-level distributed directory service – a very important component of RoCL – were presented.

The case study “TPVM over RoCL" shows that the abstractions provided by RoCL allow the rapid design and implementation of a great variety of high-level applications.

Performance values indicate that RoCL exploits efficiently the low-level communication subsystems and that multi-threaded dispatching mechanisms are now feasible.

Scalability evaluation of the directory service is still undergoing. Due to the limited number of cluster nodes available for testing, we intend to use simulation techniques.

References

1. A. Alves, A. Pina, J. Exposto, and J. Rufino. ToCL: a thread oriented communication library to interface VIA and GM low-level protocols. to appear ICCS ’03, 2003.
2. M. Beck, J. Dongarra, G. Fagg, G. A. Geist, P. Gray, J. Kohl, M. Migliardi, K. Moore, T. Moore, P. Papadopoulous, S. Scott, and V. Sunderam. HARNESS: A next generation distributed virtual machine. *Future Generation Computer Systems*, 15(5–6):571–582, 1999.
3. L. Bougé, J.-F. Méhaut, and R. Namyst. Madeleine: An Efficient and Portable Communication Interface for RPC-Based Multithreaded Environments. In *PACT ’98*, 1998.
4. Compaq Computer Corp., Intel Corporation & Microsoft Corporation. Virtual Interface Architecture Specification. http://www.vidf.org/info/04standards.html, 1997.
5. R. Espenica and P. Medeiros. Porting PVM to the VIA architecture using a fast communication library. In *PVM/MPI ’02*, 2002.

6. J. Ferrari and V. Sunderam. TPVM: Distributed Concurrent Computing with Lightweight Processes. In *HPDC '95*, 1995.
7. S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A Directory Service for Configuring High-Perf. Distributed Computations. In *HPDC '97*, 1997.
8. J.-S. Kim, K. Kim, and S.-I. Jung. SOVIA: A User-level Sockets Layer Over Virtual Interface Architecture. In *CLUSTER '01*, 2001.
9. Myricom. The GM Message Passing System. http://www.myricom.com, 2000.
10. R. Namyst and J. Méhaut. PM^2: Parallel Multithreaded Machine. A computing environment for distributed architectures. In *ParCo'95*, 1995.
11. National Energy Research Scientific Computing Center. M-VIA: A High Performance Modular VIA for Linux. http://www.nersc.gov/research/FTG/via, 2002.
12. A. Pina, V. Oliveira, C. Moreira, and A. Alves. pCoR - a Prototype for Resource Oriented Computing. In *HPC '02*, 2002.