High performance multithreaded message passing on a myrinet cluster

Albano Alves[1], António Pina[2], José Exposto[1] & José Rufino[1]
[1]*Instituto Politécnico de Bragança, Portugal.*
[2]*Universidade do Minho, Portugal.*

Abstract

The main purpose of this paper is to present the impact of pCoR thread level message passing facilities on applications running in a Myrinet cluster.

To exploit Myrinet technology we use the GM library, which provides a limited number of ports as abstractions to name communication end-points. pCoR communication layer multiplexes GM ports by using a dispatcher thread to handle messages to/from a large number of communication entities (working threads).

Our approach combines polling operations executed by multiple threads to avoid unnecessary context switching; a simple mechanism is used to engage working threads into the polling scheme. We aim to reduce the total number of polling operations required to hold message passing system performance.

1 Introduction

Low latency communication technologies and SMP workstations make possible to build clusters that attain the performance of massively parallel machines. However, software development for such systems is still a major concern because it is hard to take full advantage of hardware capabilities.

pCoR [1] communication layer focus on two main aspects: user level communication and our particular concept of distributed multithreaded applications.

1.1 User level communication systems

Nowadays operating systems provide built in communication abstractions to interface a variety of network adapters. Operating systems communication primitives inflict unacceptable delays because of excessive context switching and memory

copying. To overcome this limitation, several user level communication systems have been developed; applications are allowed to directly interface network interface cards bypassing the OS kernel. Furthermore, by means of memory registration mechanisms, data is moved from user address space to network through DMA enabling zero-copy communication.

GM [2] and BIP [3] are well-known user level communication libraries designed for high performance networking on Myrinet technology. Some VIA [4] implementations are also coming out making possible the exploitation of other communication technologies like Gigabit.

User level communication libraries provide low-level abstractions that are not suitable for application development. For instance, GM is not thread safe and imposes an event oriented model to send and receive messages and BIP does not provide any error recovery facility. Higher level libraries like MPICH-GM [5] and VIA-GM [6] include some thread support but they don't manage buffers efficiently; both copy user data into pre-registered memory before sending it.

To efficiently handle multithreading and communication, PM2 [7] and PANDA [8] use specific user level thread libraries which include the ability to manage communication. This approach limits the programmer because user level threads may block the whole application when calling certain I/O operations and user level thread libraries are not able to take full advantage of SMP systems.

1.2 Distributed multithreaded applications

Multithreading primary motivations are to exploit multi-processor machines and to hide I/O latency in order to achieve lower execution times for applications.

Although several authors have been concerned about the usability of threads in fine-grain parallelism, we intend to use threads as a structural element and for long-term computation. Complex distributed parallel applications (information retrieval systems, for example) can be designed to use a limited number of processes and a considerable number of threads. For each process, thread sets may be pre-assigned to stages [9] avoiding complex mechanisms to evaluate the minimum computation time that counterbalance the penalty of creating a new thread [10].

Distributed multithreaded applications require additional mechanisms for data exchange and synchronization between threads that execute on different cluster nodes. It is also important to have a unique set of user API primitives that can be used besides the location of a thread (local to the process, local to the machine or remote). pCoR combines shared memory and message passing over a communication layer that provides thread-to-thread communication for Myrinet clusters. Future work will provide Fast Ethernet and Gigabit support by using VIA.

2 Message passing and multithreading

Flexibility and robustness are two important characteristics we planed to include in pCoR that shape our preference for LinuxThreads and GM. LinuxThreads is part of *glibc* and guaranties thread safety for all Linux I/O primitives. GM is available

from Myricom (the Myrinet equipment manufacturer) and provides error recovery.

2.1 Port multiplexing

Low-level communication facilities, like those provided by GM, are intended for node-to-node communication. Thus few abstractions are available for the naming of communication end-points. GM allows opening up to eight ports (per NIC), which are really insufficient for multithreaded applications. Note that ports can't be shared among threads (GM is not thread safe).

pCoR multiplexes GM ports through a dispatcher thread; current pCoR implementation opens one GM port per process and launches a dispatcher thread to manage that port. The dispatcher is essential for message reception. In fact, it's easy to implement GM port sharing among threads by using a single *mutex* if we consider only message sending. Message reception is the difficult part; messages arrive asynchronously and network must be drained someway. Note that it is not practicable to block any thread on a specific GM port because it would delay sendings until a message arrival. Thus the dispatcher polls the GM port, enqueues message handlers for received messages and wakes up threads waiting for specific messages.

Polling is a CPU consuming task and therefore different strategies must be investigated in order to minimize pCoR communication layer impact on applications performance. Nevertheless it is important to note that reducing polling cycles (to free CPU for useful computation) may result in poor application performance due to degradation of communication latency (see 3.3).

pCoR includes a primitive to configure the dispatcher behaviour; it is possible to force CPU yielding after successful polls and to set the waiting time between successive polls. Furthermore the dispatcher is able to detect when the majority of the application threads are waiting for messages to increase the poll frequency. Application threads may also collaborate on message dispatching calling another special primitive that executes the dispatcher routine.

Some authors ([11], [12], [13]) propose the inclusion of polling mechanisms into the thread scheduler. Our work aims to build an efficient platform without patching or substituting the LinuxThreads scheduler.

Port multiplexing further requires a naming service to map thread identifiers into GM identifiers. pCoR accomplishes a basic naming service over TCP/IP. A thread identifier is resolved the first time a message is sent to that specific thread and the corresponding mapping is stored in a local cache.

2.2 Towards zero-copy

User level communication libraries allow zero-copy data exchanging between user address spaces from different machines. GM forces the use of *DMAble* memory blocks to hold data to be sent to other nodes. At the receiving side pre-allocated *DMAble* buffers must also be available to store incoming data.

Higher level programming abstractions must integrate those low level facilities

to avoid extra copies. However, it is not an easy task to combine the GM communication model and a particular high-level communication abstraction. Sockets-GM [14], VIA-GM and MPICH-GM, for instance, pre-allocate *DMAble* buffers at library start-up and copy application data to/from that buffers before/after sending/receiving it (two copies are made).

In addition to traditional send and receive operations, pCoR communication layer provides four extra operations to manage zero-copy communication: buffer request, buffer probe, buffer immediate release and buffer delayed release.

A buffer request returns a pointer to a *DMAble* memory block. A dynamic pool of memory blocks is maintained to avoid memory allocation and memory registration operations. Any size (s) memory blocks may be requested for sending data but pCoR will always allocate $2^n - 8$ bytes ($s \leq 2^n - 8, n \in N$) because we consider that any block may be used for future reception and GM forces receiving buffers to match this size restriction.

Programs compute new data into a *DMAble* memory block and whenever it is necessary to send that data to a remote thread no memory copy will occur. Because sending is an asynchronous operation, buffer probing is required to guarantee that data is not changed while it is moved (by using DMA) to the NIC.

Buffer release operations are intended to reduce memory waste mainly caused by receptions. In fact, at message arrival a *DMAble* block is used and it will be necessary to provide a new block (with the same size) to the GM library. If no memory blocks are available from the pool, pCoR will allocate a new one. Since pCoR receive operation returns to the user a pointer to a *DMAble* block, after reading/using the received data the program must release the corresponding buffer so that pCoR may reuse it. Delayed releases are used to notify the pCoR communication layer that it may reuse a specific buffer after the terminus of the corresponding sending operation while immediate releases are used whenever a buffer may be reused immediately.

2.3 Raw performance

Figure 1 presents raw performance obtained for message exchange between two pCoR threads. GM performance for message exchange between nodes is also presented as a basis for comparison.

Evaluation was undertaken by using two dual Intel PIII workstations (733MHz), running Linux RedHat 7.3 (kernel 2.4.18-3smp). The workstations were interconnected by Myrinet LANai9 cards connected to 64bits/66MHz PCI slots.

It is important to note that pCoR message exchange incurs in a constant penalty – $\approx$55μs. The major part of that time is consumed on thread wake-up at each communication end-point. pCoR uses condition variables to block threads until message arrival and basic LinuxThreads evaluation confirmed that a single thread wake-up (by using a condition variable) takes above 25μs on our SMP workstations. So, at least 50μs will be consumed waking-up the two application threads responsible for a message exchange.

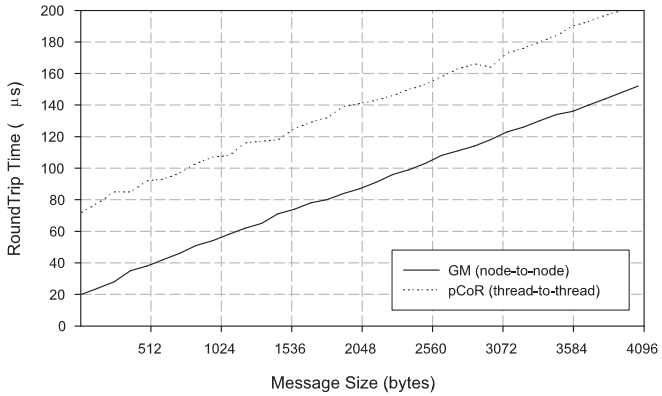

Figure 1: Raw performance for pCoR and GM.

3 Overlapping of computation and communication

Most of evaluation, tuning and comparison of message-passing systems rely on simple latency and throughput tests, still programmers are mainly concerned about the impact of communication on program computations.

Since there is not a general evaluation tool to use for distributed parallel environments testing, we developed a basic synthetic test to evaluate and tune pCoR. We also intend to use this synthetic test to compare pCoR with other systems (MPICH, PM2, PANDA and Athapascan [15]).

3.1 The synthetic test

In our test, tokens travel across cluster nodes consuming CPU cycles, every time they arrive to a node.

A single cluster node is used to produce tokens at start-up, which are sent to random destinations chosen from the remaining cluster nodes. On those nodes several threads are created by using a certain number of processes (according to test parameters). A valid destination for a token is a thread running in a specific process on a specific node.

Each token carries a random seed, a time to live and a work level. The seed and the time to live are used at each arriving node to randomly compute the next token destination. The computing time required by each token is determined by the work level. The time to live is decremented at each node and when it reaches zero the token is returned to the producer node.

The producer node computes the elapsed time between the generation of the first token and the return of the last one. By varying the number of processes at each node we can evaluate the impact of using a different number of GM ports.

By varying the number of threads per process it is possible to quantify port multiplexing overhead. To find how computation overlaps with pCoR communication we use different token work levels.

3.2 Evaluation

To evaluate pCoR implementation we ran our synthetic test using five cluster nodes with technical specifications similar to that pointed in 2.3. Four cluster nodes were used to execute multiple combinations of processes (P) and threads (Th) per node ($P \times Th \mid P \in \{1,2,4\} \wedge Th \in \{1,2,4,8\}$). The fifth cluster node was used to produce 32 tokens (Tk) with 256 bytes length, carrying a 10000 time to live (TTL) and a work level $W \in \{0,1,10\}$. A W work level is equivalent to $(W \times 145)$ μs of computation, to be consumed whenever a token arrives to a node.

For each scenario, the execution time may be estimated by the expression $[(Tk \times TTL \times W \times 145) \div max(P \times Th, 2) + Ct]$ μs, assuming that tokens will visit uniformly all nodes and considering that two processors are available per node. Communication time (Ct) could be estimated by $[Tk \times TTL \times (RoundTrip \div 2)]$ μs (using values from 2.3), but it would be to assume in advance that message transfer latency is independent from computation and multithreading. Therefore we define communication time as the time required to exchange messages plus the overhead inflicted on application execution.

Figure 2 presents communication times for all possible scenarios. Values were calculated by subtracting token processing times from synthetic test execution time (STt): $Ct = STt - [(Tk \times TTL \times W \times 145) \div max(P \times Th, 2)]$.

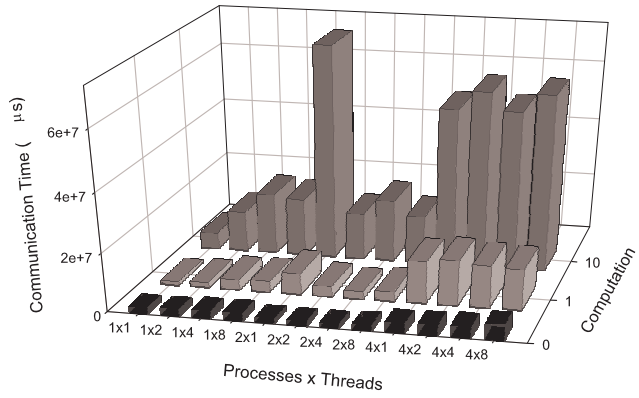

Figure 2: Test results for different scenarios.

As expected communication time increases whenever we force a higher work level per token. On the one hand, the dispatching system consumes CPU cycles

reducing the computing node power. On the other hand, computing cycles required to process each token delay the dispatcher raising message latency.

Another relevant conclusion is that thread context switching has a major impact on application performance as we increase the number of application threads. In fact, the OS thread scheduler assigns less CPU cycles to the pCoR dispatcher whenever the total number of threads increases.

It is also important to note that when we used two processes per node and one thread per process poor performance was achieved. Additional testing is required to come to a precise conclusion but we suspect that some drawback exist on GM NIC multiplexing.

3.3 Polling strategies

As we mentioned in 2.1, pCoR allows different polling strategies. To evaluate the impact of polling we run the synthetic test for a specific scenario (Tokens=32, TTL=10000, Processes=1, Threads=8, Work=10) using different polling strategies.

A particular polling strategy is defined by a tuple $D/A \mid D \in \{p, y, l\} \wedge A \in \{0, 5, 10, 100, 1000\}$). D characterizes the thread dispatcher behaviour: p and y means respectively that the thread dispatcher will pause for 20μs or yield the processor after polling while l is used to point that the dispatcher enter an infinite loop polling the GM port at maximum rate . A quantifies the number of polls performed during token processing, in order to help the dispatcher thread. Application threads accomplish polling by calling the pCoR dispatcher routine as mentioned in 2.1.

Figure 3 presents total execution time and average number of polls (per node) for each run.

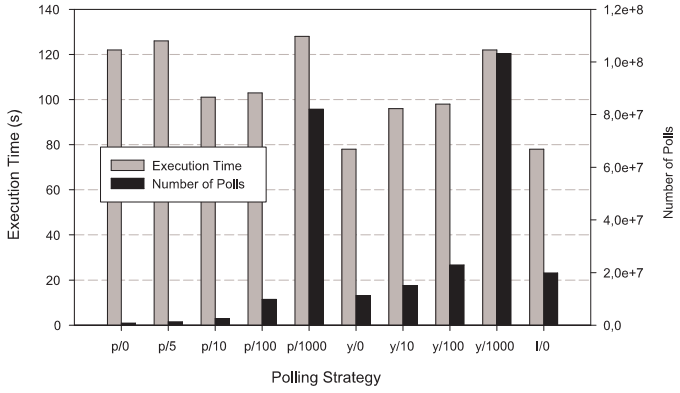

Figure 3: Impact of different polling strategies.

Note that lower polling rates will result in poor application performance because thread-to-thread latency will increase significantly. On the other hand, excessive polling will drop computing power resulting in poor performance too.

It is important to note that strategies $y/0$ and $l/0$ produced similar performance results despite of the different number of polls. Although yielding the processor after polling the GM port results in a lower number of polls, we must take into account that yielding the processor consumes CPU cycles.

We should also emphasize that the number of polls is not proportional to the final application execution time. Strategies $p/10$, $p/100$ and $y/0$ are fair examples: similar execution times and different number of polls or vice versa may occur because beyond the total number of polls it is important to know when to poll.

For applications that exchange few messages, to pause the dispatcher and engage application threads on port polling will result in lower CPU utilisation and will increase application performance. Our synthetic test requires 320000 messages to be sent in a short period of time and obviously it is not a good example to show this advantage.

4 Thread libraries

Available thread libraries rely on one of three models: 1:1, N:1 and N:M. LinuxThreads uses the 1:1 approach, which means that a user thread matches a kernel thread. This allows taking full advantage from SMP workstations and guarantees total safety when calling I/O primitives. As a disadvantage we must point out that LinuxThreads achieves poor performance on context switching and thread synchronisation.

NGPT (Next Generation Posix Threading) [16] is an emerging thread library (for Linux), which relies on the N:M model. It will be possible to execute applications compiled for LinuxThreads and new applications will able to combine user threads and kernel threads. Context switching and synchronisation for user threads is significantly faster but NGPT developers sustain that NGPT kernel threads outperform those from LinuxThreads.

4.1 NGPT vs LinuxThreads

Although NGPT is still under development, we designed a simple multithreading evaluation test to compare LinuxThreads and a NGPT beta release. It would be obviously more reasonable to evaluate pCoR by using each of the thread libraries, but it is yet not possible to use the GM driver and the NGPT thread library because NGPT requires a kernel version still not supported by GM.

The evaluation test consists of one producer thread generating events and several consumer threads handling those events. This way we simulate the pCoR communication environment where a single library thread dispatches messages to several application threads.

The producer generates 100000 events and delivers them randomly to the available consumers. After delivering an event the producer enters a loop in order to

produce a 10 μs delay. Consumers wait for events by blocking themselves using condition variables.

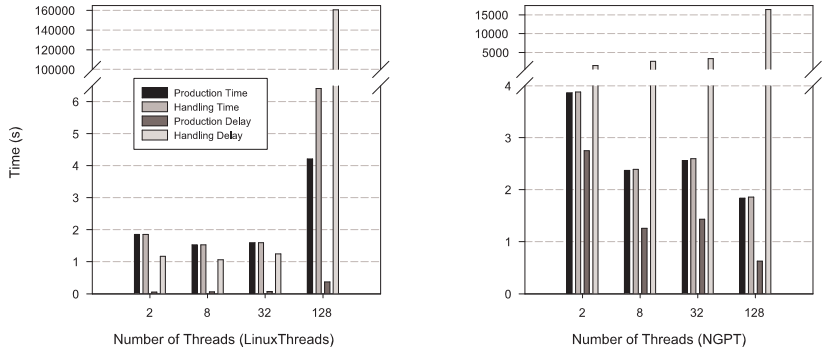

Figure 4: Event handling using LinuxThreads and NGPT.

Figure 4 presents total production and handling times along with production and handling delays. Production and handling times refer respectively to the time required by the producer to generate all the events and to the time required by all consumers to handle those events. Production delays are calculated by subtracting the instant an event should be generated from the instant it is effectively generated (each event should be generated 10 μs after the deliver of the previous one). Handling delays are calculated by subtracting the instant the event was generated (the produced time-stamps each event) from the instant it is handled.

Our test shows that by using NGPT it is possible to achieve better production and handling times whenever we use a large number of threads. Production and handling delays are significantly higher than those achieved by using LinuxThreads, mainly if few consumers exist. However, since we have only used NGPT kernel level threads we consider NGPT a promising platform to manage pCoR threads.

5 Conclusions

The communication layer developed to pCoR allows message passing between threads residing on any node of a Myrinet cluster. Multiple threads may collaborate to accomplish a single message exchange in order to minimize the impact of port multiplexing. Zero-copy facilities provided by the low-level communication library were integrated with higher level programming abstractions to guarantee low latency and high throughput.

We have developed a basic synthetic test to evaluate and tune pCoR, since there is not a general tool for the purpose of evaluating distributed parallel platforms. The synthetic test proved to be very useful and we intend to use it to compare pCoR with other systems.

Thread context switching imposes significant overhead on thread-to-thread message passing. Preliminary testing points that by using NGPT it will be possible to reduce pCoR communication layer overhead.

References

[1] Pina, A., Oliveira, V., Moreira, C. & Alves, A. pCoR - a Prototype for Resource Oriented Computing. submitted to HPC '02, 2002.

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

[3] Prylli, L. & Tourancheau, B. BIP: a new protocol designed for high speed performance networking on Myrinet. Workshop PC-NOW, IPPS/SPDP98, 1998.

[4] Compaq Computer Corp., Intel Corporationnand & Microsoft Corporation. Virtual Interface Architecture Specification, 1997.

[5] Gropp, W. & Lusk, E. *Installation and User's Guide to MPICH, a Portable Implementation of MPI*, 2001.

[6] Myricom. *VI-GM: Virtual Interface on Myrinet*, 2002.

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

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

[9] Welsh, M., Culler, D. & Brewer, E. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Eighteeth Symposyum on Operating Systems Principles (SOSP-18)*, 2001.

[10] Cohen, W., Patel, C. & Seshagiri, A. Cost of User and Kernel Level Threads Operations on Linux, 1998.

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

[12] Hansen, J. & Jul, E. Latency Reduction using a Polling Scheduler. In *Second Workshop on Cluster-Based Computing*, pages 27–31. ACM, 2000.

[13] Danjean, V., Namyst, R. & Russel, R. Linux Kernel Activations to Support Multithreading. In *18th Interbational Conference on Applied Informatics (AI 2000)*, 2000.

[14] Myricom. *Sockets-GM*, 2002.

[15] Briat, J., Ginzburg, I. & Pasin, M. *Athapascan-0 User Manual*, 1998.

[16] Abt, B., Desai, S., Howell, D. & McCracken, D. Next Generation POSIC Threading: Moving Linux to the Enterprise, 2002.