Reducing the Overhead of Intra-Node Communication in Clusters of SMPs

Sascha Hunold and Thomas Rauber

Department of Mathematics, Physics and Computer Science
University of Bayreuth, Germany
{hunold,rauber}@uni-bayreuth.de

Abstract. This article presents the C++ library vShark which reduces the intra-node communication overhead of parallel programs on clusters of SMPs. The library is built on top of message-passing libraries like MPI to provide thread-safe communication but most importantly, to improve the communication between threads within one SMP node. vShark uses a modular but transparent design which makes it independent of specific communication libraries. Thus, different subsystems such as MPI, CORBA, or PVM could also be used for low-level communication. We present an implementation of vShark based on MPI and the POSIX thread library, and show that the efficient intra-node communication of vShark improves the performance of parallel algorithms.
Keywords: clusters of SMPs, parallel programming models, message passing between threads

1 Introduction

Clusters of SMPs (Symmetric Multiprocessors) have become very popular in high performance computing (HPC). Due to the huge number of different cluster systems, the message passing libraries such as MPICH or LAM are usually not machine optimized. One disadvantage of MPI (Message Passing Interface) libraries is their low performance for intra-node communication. The communication on a single SMP node is either done via shared memory (system calls like `shmget`) or socket-based. Intra-node communication via sockets or shared memory at operating system level is more expensive than copying data directly between lightweight threads. vShark provides an effective realization of the communication requirements of an application that can be adapted to the memory and network system of the parallel or distributed platform without help from the programmer. In particular, vShark reduces the overhead of intra-node communication in clusters of SMPs by introducing a thread-based architecture. Instead of starting a number of processes on SMP nodes, the vShark system starts the same number of threads. Since those threads live in the same address space of the same process, communication between them is much faster than going through the communication stack of the message-passing library. The communication between physically distributed threads in vShark is performed by a separate communication thread. Each physical SMP node has exactly one communicator thread which handles communication requests from local worker-threads and polls for requests from remote communicator threads.

The next section gives a short overview of the vShark framework. The rest of the paper describes the C++ implementation of the vShark interface using the MPI standard

and the POSIX thread library. The article introduces a simple but effective communication protocol to ensure thread-safe communication between worker-threads. We also evaluate the performance of vShark and present experimental results.

2 Programming model of vShark

The vShark library is an improved message-passing framework for distributed memory machines. Thus, the programming model is the same as for common message-passing environments like MPI, i.e. explicit messages-passing between participating processes is required.

The vShark library consists of different layers to provide maximum flexibility. Parallel programs based on vShark have a common interface to the top layer runtime interface. The layer below is the transportation layer of the vShark runtime. The transportation layer binds the vShark runtime to a particular communication device like MPI or PVM. The programmer does not have access to the communication layer directly. Instead, he must use abstract functions of the vShark runtime library to send or receive data.

In this article we can only give a coarse introduction of the system. vShark provides a message-passing API which is similar to MPI. It contains methods for sending and receiving messages like `int send(Envelope *env)`, and it also contains entities such as `VSharkGroup` which is the logical equivalent to an MPI communicator. The code below is an example of how a processor would send its own rank to processor 1 in vShark.

```
Runtime& re = get_runtime();                    // get vShark runtime handle
VSharkGroup *group = re.get_group();            // handle to world communicator
Message *msg = new IntMessage(&rank, 1, 0);     // int of length 1 and with tag 0
group->send(group->create_envelope(msg, rank, 1)); // blocking send
```

Fig. 1. Sending an integer message in vShark.

3 vShark implementation with MPI and POSIX threads

vShark can be implemented on top of different communication libraries. In this section, we describe a vShark implementation based on MPI and the POSIX thread library.

General communication scheme The MPI standard does not guarantee thread-safety. Therefore, the vShark driver for MPI has to ensure thread-safe communication between worker threads. Thread-safe communication in this context is achieved when only one thread per node is transferring data at a time. Several solutions were proposed in literature, e.g. protecting all MPI calls with locks to ensure mutual exclusion, see [4] for a detailed analysis. Another solution for thread-safe communication is an auxiliary communicator thread that manages communication requests. This thread is the only one with access to the MPI layer. vShark uses such a communicator thread. A distinct communicator not only ensures thread-safe communication, but more importantly, it also allows us to change the communication path (channel) at runtime (shared-memory or

socket-based). When a virtual processors (worker thread) wants to send or receive data, it appends a request to the communicator's request queue. We explicitely indicate that vShark does not copy messages into a separate buffer. Instead, the virtual processor passes a memory reference to the communicator. After the data transfer is completed, the communicator signals the virtual processors that the requests have been fulfilled.

Message transfer protocol There are two performance-critical decisions to make. The first is, how and when communication between two communicators takes place, i.e. how often does the communicator need to poll for inter-node requests. Secondly, does the system support buffering of messages?

vShark does not buffer incoming messages to reduce the memory requirements and to avoid deadlocks through insufficient free memory. Such a scenario may occur if a communicator thread constantly polls for incoming messages and receives a large amount of incoming data within a short time interval. However, the time at which the data is actually requested is unknown, and so, the message has to be kept in the buffer. Especially in numerical applications with messages of hundreds of megabytes the fast growing buffer would quickly exceed the memory limit.

vShark uses a communication protocol to avoid deadlocks and extra memory requirements. The transfer of messages is always initiated by the the sending communicator. The communicator sends a request message to the node where the receiver resides. This message contains the id of the virtual processor of the sender and the receiver. The communicator of the receiver checks its local queue if the corresponding virtual processor has already requested this data. If so, the communicator sends an acknowledgment-message (ACK) to the initiator and immediately starts receiving data (`MPI_Irecv`) into the message buffer of the virtual processor. If there is no such request, the communicator enqueues the request in a waiting list. When a virtual processor later dispatches the matching receive request, the ACK will immediately be sent to the initiator. In order to find the corresponding request to each ACK and vice versa, the ACK message also contains the identifiers of the sending and receiving virtual processors. This protocol has two basic advantages: (1) No additional message buffering is required. (2) The initiating communicator can select which message is sent first. That makes it possible to reschedule and optimize the message transfer respecting the message-passing constraints such as order and fairness (subsequent messages may not overtake each other).

Realization of the communicator thread As discussed before, the communicator constantly polls for incoming requests. The central performance question is, when and for how long the communicator thread will be suspended. This sleep time must be short enough to guarantee quick message delivery, but also long enough that worker threads can get the CPU and perform their tasks. In case of shared-memory, we could use a consumer / producer model. The consumer would be suspended until produced items are available for consumption and so, it would not consume CPU time. Unfortunately, we cannot apply this model in a distributed memory environment. Thus, active waiting for incoming messages is necessary which may consume CPU time. Since we want to minimize this wait overhead, we introduce a sleep time for the communicator. The communicator sleeps for the given amount of time when all local queues are empty and no remote request has yet been received. We will see that this timeout parameter is very

performance-critical. The timeout settings (minimum, maximum, default) of the MPI driver can be changed in a configuration file called `vshark_mpi.conf`.

Running vShark programs over MPI An MPI-based vShark program can be started by calling `mpirun` on each participating node. The runtime system of vShark reads the node configuration file `vshark.conf`. For compatibility reasons, this file has the same syntax as the machine configuration files of MPICH (`node:#processors`). According to the number of processors specified in the file, the vShark runtime starts the virtual processors. After the runtime is loaded on each node, the actual vShark program is passed to the virtual processors which then start to execute the program.

4 Experimental results of the MPI version of vShark

For a performance comparison of vShark with traditional MPI programs, we ran several benchmarks from the ParkBench collection [9]. The original ParkBench code is written in Fortran 77. We ported the benchmarks to C++ and replaced MPI calls with the corresponding vShark function.

In the diagrams that follow, "parkbench" denotes the results of the original benchmark and vShark stands for the rewritten benchmark. The range $(x-y)$ after the vShark label denotes the chosen minimum and the maximum timeout of the communicator, e.g. for $1-10$ the communicator waits at least $1\ ms$ and at most $10\ ms$ when all queues are empty.

COMMS1 benchmark The COMMS1 benchmark is a so called ping-pong benchmark and measures the time a message is transferred between two nodes back and forth, i.e. the master processor sends a message of variable length to a slave processor that immediately returns the message after receiving it.

Fig. 2 (left) presents the throughput results for the intra-node communication of vShark and ParkBench. It can be observed that the communication between two MPI processes (original ParkBench) is fast for smaller messages. When the message size increases, vShark's thread-based copying significantly boosts the performance. For a message size of about 20.000 bytes, the throughput of vShark becomes clearly superior to MPI.

COMMS3 benchmark The website `www.top500.org` characterizes the benchmark as follows: each processor of a p-processor system sends a message of length n to the other $(p-1)$ processors. Each processor then waits to receive the $(p-1)$ messages directed at it. The timing of this generalized ping-pong ends when all messages have been successfully received by all processors; although the process will be repeated many times to obtain an accurate measurement, and the overall time will be divided by the number of repeats. Figure 2 (right) compares the bandwidth which was measured for the MPI version of COMMS3 and the vShark version. When utilizing four processors, the bandwidth achieved by vShark is slightly lower than the MPI version. Yet, when the message length is larger than 50.000 bytes vShark is as fast as the original ParkBench. Due to the additional communication protocol, the bandwidth of vShark decreases for a larger number of processors.

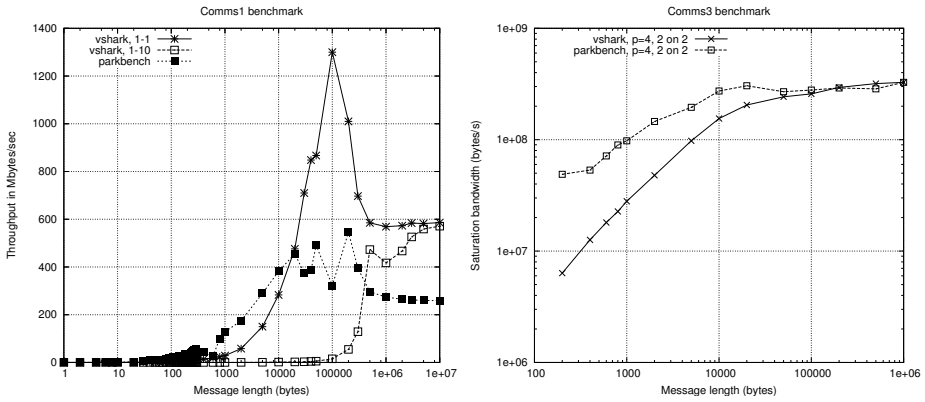

Fig. 2. left: throughput measured with the COMMS1 benchmark (intra-node, SMP)
right: two virtual vShark processors on two SMP nodes compared to two MPI processes on two SMP nodes. system: dual Xeon cluster, SCI network, ScaMPI.

Testing with a real application We examine the real-world application performance on the basis of tpMM (task parallel matrix multiplication). The tpMM algorithm uses a hierarchy of multiprocessor groups where it is assumed that matrix A is decomposed into p blocks of rows and B into p blocks of columns, where p denotes the number of processors. tpMM recursively updates matrix panels to compute the result matrix $C = A \times B$, see [6] for a more detailed description of tpMM and [7] for an overview of how tpMM can be used as a building block in multi-level matrix multiplication algorithms.

The runtime results for tpMM on vShark and on MPICH are shown in Figure 3 (left). We can see that tpMM running on vShark outperforms the C/MPI version (note the logarithmic scale). This algorithm benefits from the vShark runtime since most of the communication required happens on an SMP node.

In order to evaluate the performance of vShark on larger SMP nodes, tpMM was further tested on a four-way Xeon (2.0 GHz) running Linux and MPICH 1.2.5. Figure 3 (right) compares the MFLOPS per processor of the vShark-based and the MPICH-based versions of tpMM. The MPICH results include statistics for the P4 device (shared memory enabled) as well as for the VMI device. On a multi-way SMP machine, vShark clearly outperforms the MPICH versions, either using the VMI or the P4 driver.

5 Related work

The combination of message passing in a multi-threaded environment and its advantages has already been examined and published. For example, Sun Microsystems offers thread-safe MPI libraries for Solaris [13] where threads can concurrently call MPI functions but may only refer to processes as senders or recipients. Multi-threaded approaches to MPICH have been discussed in [11]. The article [4] describes how to use threads in an MPI environment efficiently to improve the performance of irregular algorithms on distributed systems. In general, there are two approaches for exploiting threads in distributed systems. One way is to create a virtual shared model of the parallel system.

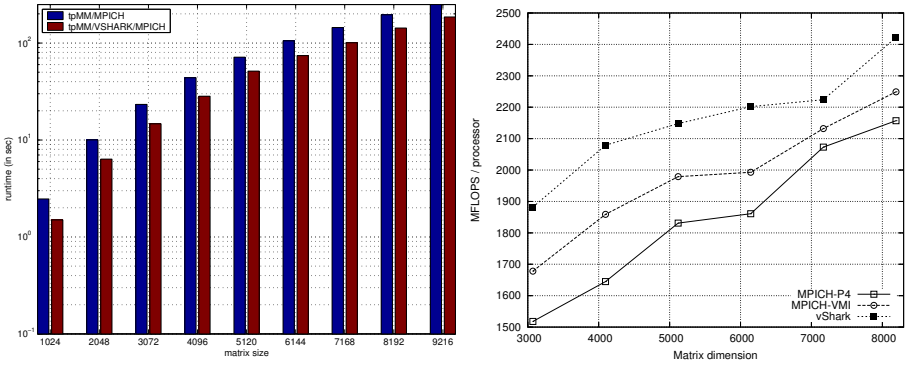

Fig. 3. left: comparison of the performance of tpMM running on vShark and directly on MPI (8 virtual processors, 2 thread per node), system: dual Xeon cluster, MPICH 1.2.5.2 (`-with-comm= shared`).
right: tpMM performance with MPICH-P4, MPICH-VMI, and vShark, system: 4-way Xeon.

Since the programmer sees only one big memory, the complexity of writing parallel program decreases because explicit message passing is omitted. MuPC is an example of such programming language [12]. Another approach is to extend the POSIX thread model and to add message passing capabilities to each thread [3]. In [1] the authors proposed a thread-only implementation of MPI and it aims at the development stage of program where tests are performed on a single machine. The work in [10, 14] goes one step further and rewrites parts of MPICH to shift the original process-only model to a thread-only model. The disadvantage of these approaches is the dependency on the operating system and the MPICH version. Another multi-threaded MPI implementation is called AMPI and has been discussed in [5]. AMPI uses the same notation of virtual processors as vShark. Each virtual processor is a lightweight user-thread and has its own private memory. AMPI optimizes the mapping of virtual processors to real processors. The objective of AMPI is to reduce the complexity of writing parallel programs for systems where the number of processors differs from the number of processors that the algorithms require. [2] introduces TPVM as a multi-threaded version of PVM. Similar to vShark, TPVM uses threads as units of parallelism and the communication between threads is done via explicit message passing with a unique thread id. Since TPVM is a modified version of PVM, it is restricted to the PVM library and the operating systems to which it has been ported. The Virtual Machine Interface (VMI) is also equipped with the support for multiple communication interconnects including shared memory, TCP/IP, Myrinet [8]. In contrast to vShark, VMI is a middleware layer between MPI and the network device drivers.

6 Conclusions

We have presented the `C++` library vShark which is built upon message passing and thread libraries. Despite having a distributed programming model, communication between virtual processors which are implemented as lightweight threads is done without invoking external library functions or operating system routines. The experimen-

tal results have shown that parallel programs that use vShark as communication layer can lead to significant performance gains when many intra-node communications are performed. The main advantage of vShark is its flexibility through the object-oriented design and the placement on top of message passing libraries. Thus, porting vShark programs to different architectures is easy since it only requires a single vShark driver for a new communication interface like MPI or PVM. Since there is already an MPI 1.1 binding available, vShark will work with any MPI compliant library.

References

1. Erik D. Demaine. A Threads-Only MPI Implementation for the Development of Parallel Programs. In *Proc. of the 11th International Symposium on High Performance Computing Systems (HPCS'97)*, pages 153–163, Winnipeg, Manitoba, Canada, July 1997.
2. Adam Ferrari and V. S. Sunderam. Multiparadigm Distributed Computing with TPVM. *Concurrency: Practice and Experience*, 10(3):199–228, 1998.
3. Matthew Haines, David Cronk, and Piyush Mehrotra. On the Design of Chant: A Talking Threads Package. In *Proc. of the 1994 conference on Supercomputing*, pages 350–359. IEEE Computer Society Press, 1994.
4. Judith Hippold and Gudula Rünger. A Communication API for Implementing Irregular Algorithms on SMP Clusters. In *Proc. of the 10th EuroPVM/MPI 2003*, LNCS 2840, pages 455–463, Berlin Heidelberg, 2003. Springer.
5. Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive MPI. In *Proc. of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, LNCS 2958, pages 306–322, College Station, Texas, October 2003. Springer.
6. Sascha Hunold, Thomas Rauber, and Gudula Rünger. Hierarchical Matrix-Matrix Multiplication based on Multiprocessor Tasks. In *Proc. of the International Conference on Computational Science ICCS 2004, Part II*, LNCS 3037, pages 1–8. Springer, 2004.
7. Sascha Hunold, Thomas Rauber, and Gudula Rünger. Multilevel Hierarchical Matrix Multiplication on Clusters. In *Proc. of the 18th Annual ACM International Conference on Supercomputing, ICS'04*, pages 136–145, June 2004.
8. Scott Pakin and Avneesh Pant. VMI 2.0: A Dynamically Reconfigurable Messaging Layer for Availability, Usability, and Management. In *The 8th International Symposium on High Performance Computer Architecture (HPCA-8), Workshop on Novel Uses of System Area Networks (SAN-1)*, Cambridge, Massachusetts, February 2, 2002.
9. PARKBENCH Committee/Assembled by R. Hockney (Chairman) and M. Berry (Secretary). PARKBENCH report: Public international benchmarks for parallel computers. *Scientific Programming*, 3(2):101–146, Summer 1994.
10. Jari Porras, Pentti Huttunen, and Jouni Ikonen. The Effect of the 2^{nd} Generation Clusters: Changes in the Parallel Programming Paradigms. In *Proc. of the International Conference on Computational Science ICCS 2004, Part III*, LNCS 3037, pages 10–17. Springer, 2004.
11. Boris V. Protopopov and Anthony Skjellum. A Multithreaded Message Passing Interface (MPI) Architecture: Performance and Program Issues. *Journal of Parallel and Distributed Computing*, 61(4):449–466, 2001.
12. J. Savant and S. Seidel. MuPC: A Run Time System for Unified Parallel C. Technical report, Department of Computer Science, Michigan Technological University, September 2002.
13. Sun Microsystems Computer Company. Sun MPI 4.1 Programming and Reference Guide, March 2000.
14. Hong Tang and Tao Yang. Optimizing Threaded MPI Execution on SMP Clusters. In *Proc. of the 15th International Conference on Supercomputing*, pages 381–392. ACM Press, 2001.