Implementing a Hardware-Based Barrier in Open MPI

- A Case Study -

Torsten Hoefler[1], Jeffrey M. Squyres[2], Torsten Mehlan[1]
Frank Mietke[1] and Wolfgang Rehm[1]

[1]Technical University of Chemnitz
Dept. of Computer Science
Chair of Computer Architecture
Chemnitz, 09107 GERMANY
{htor,tome,mief,rehm}@cs.tu-chemnitz.de

[2]Open Systems Laboratory
Indiana University
501 N. Morton Street
Bloomington, IN 47404 USA
jsquyres@open-mpi.org

April 15, 2006

Abstract

Open MPI is a recent open source development project which combines features of different MPI implementations. These features include fault tolerance, multi network support, grid support and a component architecture which ensures extensibility. The TUC Hardware Barrier is a special purpose low latency barrier network based on commodity hardware. We show that the Open MPI collective framework can easily be extended to support the special purpose collective hardware without any changes in Open MPI itself.

1 Introduction

Many different MPI libraries are available to the user and each implementation has its strengths and weaknesses. Many of them are based on the reference implementa-

tion MPICH [8] or its successor for MPI-2 MPICH2. Several MPI implementations offer one specific functionality in their code-base, e.g. LAM/MPI offers an easily extensible framework, FT-MPI [4] offers fault tolerance, MPICH-G2 [7] grid support, and LA-MPI [6] offers multiple network device support, but there is no MPI available which combines all of these aspects. Open MPI is intended to fill this gap and combine different ideas into a single and extensible implementation (compare [5]). Many developers with a MPI background combine their knowledge to create this new framework and to add some remarkable features to Open MPI. These developers came from different directions and want to make a new design which is free from old architectural constraints. FT-MPI [4] offers fault tolerance and design for changing environments, LA-MPI [6] offers multi-device support, LAM/MPI [3] offers the framework and PACX-MPI the multi-cluster and grid support. Open MPI started in January 2004 and was written from scratch to overcome several constraints and fit the needs of all members. Roughly two years later, version 1.0 was presented to the public.

1.1 Short Introduction to Open MPI

Regarding to Gabriel et. al. in [5], the main goals of Open MPI are to keep the code open and extensible, to support MPI-2 with the maximum thread level (MPI_THREAD_MULTIPLE) for all widely deployed communication networks. Additionally, features like message fragmentation and striping across multiple interfaces as well as fault recovery and tolerance, both totally transparent from the applications point of view, will be included in the implementation. The useability of the Run-Time-Environment (RTE) should be as comfortable as possible to support the user for execution his MPI programs. Additional features can be enabled optionally, such as data integrity check, to recognize errors on internal data busses.

To reach all this goals stated above, Open MPI uses a special architecture, called Modular Component Architecture compareable to the Common Component Architecture (CCA), described in [1]. This architecture offers a framework to implement every needed functionality for MPI in a modular manner. The necessary functional domains are derived from the goals and equipped with a clear interface as a framework inside Open MPI. Each framework can host multiple modules which implement the necessary functionality to serve the interface. Each module could support different hardware or implement different algorithms to enable the user or the framework to choose a special module for each run without recompiling or relinking the code.

The modular structure is shown in Figure 1.

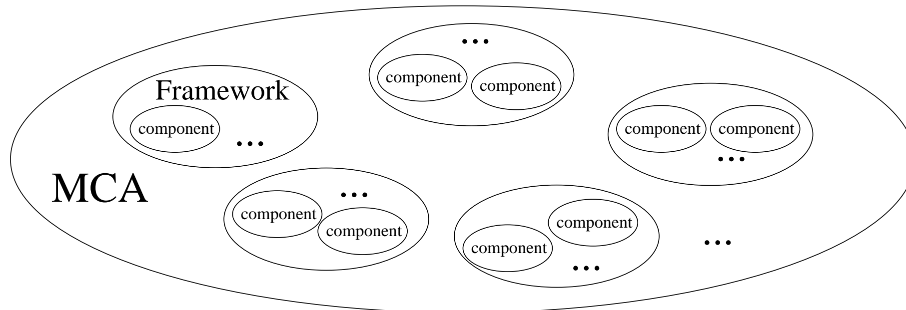

Figure 1: Open MPI Architecture

The three entities MCA, framework and modules are explained in the following. The MCA offers a link between the user, who is able to pass command line arguments or MCA parameters to it, and the modules which receive the parameters. Each framework is dedicated to a special task (functional domain) and discovers, loads, unloads and uses all modules which implement its interface. The framework performs also the module selection, which can be influenced by user (MCA) parameters. Source code for modules is supported and configured/built by the MCA as well as binary modules which are loaded by the framework via the GNU Libltdl. Current frameworks include the Byte Transport Layer (BTL) which acts as a data transport driver for each network, the BTL Management Layer which manages BTL modules and the Point-to-Point Management Layer which performs higher level tasks as reliability, scheduling or bookkeeping of messages. All these frameworks are directly related to the transport of point-to-point messages (e.g. MPI_SEND, MPI_RECV). Other frameworks, such as the topology framework (TOPO), the Parallel IO framework (IO) offer additional functionality for other MPI related domains. Helper frameworks, such as Rcache or MPool are available to BTL implementers to support the pinned management of registered/pinned memory.

The collective framework (COLL) was used to implement a module which offers the hardware barrier support. The layering and the interaction between the different frameworks in the case of InfiniBand [10] communication is shown in Figure 2. The framework is specially designed for HPC and optimized in all critical paths. Thus, the overhead introduced compared to a static structure is less than 1% (cmp. [2]).

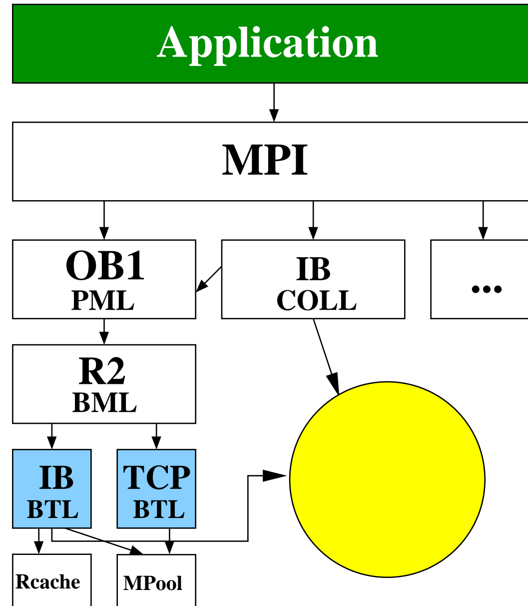

Figure 2: Interaction of Frameworks inside the MCA

2 The TUC Hardware Barrier

The TUC Hardware Barrier is a self-made barrier hardware based on commodity hardware. The standard parallel port is used for communication between the nodes and a central hardware controls all nodes. A schema of the pin-out of the parallel port is given in Figure 3. The current prototype supports only a single barrier operation between all nodes, thus it is only usable for the global communicator MPI_COMM_WORLD. Each node has a single inbound and outbound channel which can have the binary state '0' or '1' and is connected to the barrier hardware. The prototypical barrier hardware consists of a ALTERA UP1 FPGA (shown in Figure 4) and is able to control 60 nodes. The FPGA implements a relatively easy Finite State Machine with two states regarding to the two states of the input wire of each node, '0' or '1'. A transition between these two states is performed, when either all output wires of all nodes are set to '1' or all output wires are set to '0'. The graph representing the state transitions is shown in Figure 5. When a single node reaches its MPI_BARRIER call, it reads the input wire, toggles the read bit and writes it to the output wire. Then, the node waits until the input is toggled by the barrier hardware, and leaves the barrier call (all nodes toggled their bit).

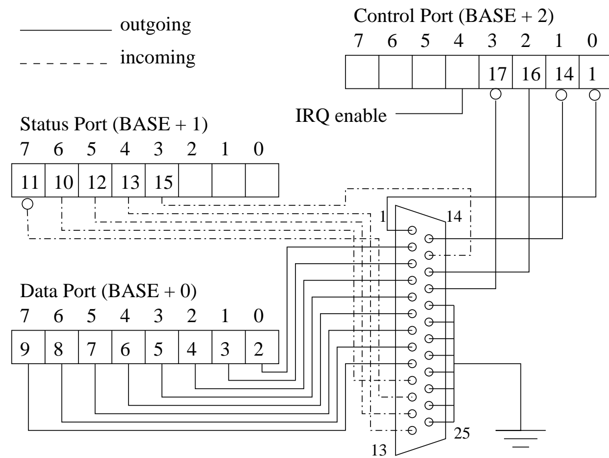

Figure 3: Parallel Port Pin-out

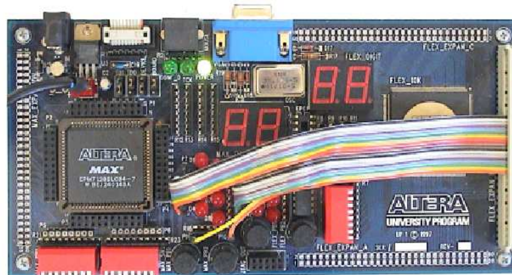

Figure 4: The prototypical Hardware

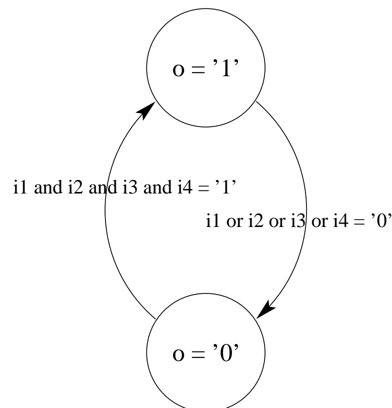

Figure 5: The implemented Finite State Machine

3 Implementing MPI_BARRIER

The barrier is implemented inside the COLL framework, which is originally introduced by Squyres et. al. in [9]. The framework finds and loads all components and queries each component if it wants to run on this machine (the component is able to check for specific hardware or other prerequisites). During the creation of a new communicator, the framework initializes all runnable modules. Each module can test for all its prerequisites on this specific communicator and return a priority to the framework. This selects the best one according to MCA parameters or priorities returned by the modules. When a communicator is destroyed, the framework calls uninitialize functions of the active modules that all resources can be freed. In particular, each COLL component implements three functions:

`collm_init_query()` Which is invoked during MPI_INIT and tests if all prerequisites are available,

`collm_comm_query()` Which is invoked for each new communicator and

`collm_comm_unquery()` Which instructs the component to free all used structures for the communicator.

Once, a component is selected for a specific communicator, it is called module. There are two basic functions for each module, coll_module_init() which can be used to initialize structures for each communicator and coll_module_finalize() which should free all used resources. Additionally, a module can implement selected or all collective functions. All collective functions which are not implemented are performed by a reference implementation based on point-to-point messages.

Our hwbarr component utilizes the component functions in the following way (note that OMPI_SUCCESS indicates the successful completion of a function to the framework):

`collm_init_query()` Checks for the appropriate access rights (IN,OUT CPU calls) and returns OMPI_SUCCESS if they are available

`collm_comm_query()` Returns a priority if the communicator is MPI_COMM_WORLD, or refuses to run otherwise

`collm_comm_unquery()` Returns OMPI_SUCCESS

The module functions are used as stated in the following:

`coll_module_init()` Returns OMPI_SUCCESS

`coll_module_finalize()` Returns OMPI_SUCCESS

`coll_barrier()` Performs the barrier call as described in section 2 by reading the input, writing the toggled value to the output and waiting until the input also toggles

Other collective functions have not been implemented and the framework uses the standard implementation if they are called.

4 Conclusions

We showed that it is easy to add support for a special hardware to support collective operations in Open MPI. The application or Open MPI installation has neither to be recompiled nor to be relinked, the module has to implement the interface functions and has to be copied to the right directory in the library path. The architecture of Open MPI adds also nearly no overhead which makes the collective framework very attractive to programmers of collective operations. The TUC Hardware Barrier and the according COLL component has to be extended to support multiple communicators and allow non-root processes to use the hardware.

References

[1] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott R. Kohn, Lois McInnes, Steve R. Parker, and Brent A. Smolinski. Toward a common component architecture for high-performance scientific computing. In *HPDC*, 1999.

[2] B. Barrett, J. M. Squyres, A. Lumsdaine, R. L. Graham, and G. Bosilca. Analysis of the Component Architecture Overhead in Open MPI. In *Proceedings, 12th European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.

[3] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[4] G. Fagg and J. Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *Lecture Notes in Computer Science: Proceedings of EuroPVM-MPI 2000*, volume 1908, pages 346–353. Springer Verlag, 2000, 2000.

[5] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004.

[6] Richard L. Graham, Sung-Eun Choi, David J. Daniel, Nehal N. Desai, Ronald G. Minnich, Craig E. Rasmussen, L. Dean Risinger, and Mitchel W. Sukalski. A network-failure-tolerant message-passing system for terascale clusters. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 77–83, New York, NY, USA, 2002. ACM Press.

[7] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5):551–563, 2003.

[8] MPICH2 Developers. http://www-unix.mcs.anl.gov/mpi/mpich2/.

[9] Jeffrey M. Squyres and Andrew Lumsdaine. The Component Architecture of Open MPI: Enabling Third-Party Collective Algorithms. In *Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, St. Malo, France, July 2004.

[10] The InfiniBand Trade Association. *Infiniband Architecture Specification Volume 1, Release 1.2*. InfiniBand Trade Association, 2003.