Adding Low-Cost Hardware Barrier Support to Small Commodity Clusters

Torsten Hoefler, Torsten Mehlan, Frank Mietke and Wolfgang Rehm

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

Abstract: The performance of the barrier operation can be crucial for many parallel codes. Especially distributed shared memory systems have to synchronize frequently to ensure the proper ordering of memory accesses. The barrier operation is often performed by issuing point-to-point messages and the best algorithm scales with $O(log_2 P \cdot L)$ in the LogP model. We propose a cheap hardware extension which is able to perform the task of synchronization in nearly constant time and implement a driver inside the Open MPI framework to speedup the MPI_Barrier() call. We test our implementation with the parallel version of Abinit resulting in an MPI overhead decrease of nearly 32%.

1 Introduction

The barrier operation can be important for different applications. Especially DSM (Distributed Shared Memory) systems perform the barrier operation frequently to ensure the proper ordering of remote memory accesses. The overall latency of the barrier essentially consists of two parts, the synchronization time and the process skew. The process skew starts to increase after each synchronization due to operating system influence and load imbalance. It is defined as the time difference between the first and the last process which calls the barrier function. This skew occurs due to the application itself (unfavorable load distribution) or the runtime environment (e.g. operating system deamons running on the nodes) and is not addressed in this paper. The second part, the synchronization time can be measured when all nodes reach their barrier at the same time. It is the time that is needed to communicate the "barrier reached" state to all other nodes. Traditional barrier implementations for distributed memory systems use explicit messages (point-to-point operations) to synchronize among each other. The fastest algorithm for this purpose is the Dissemination algorithm [HFM88] and its optimality within the LogP [CKP+93] model has been proven in [HMMR04]. These theoretical considerations have been verified with practical measurements in [HCM+05] which leads us to the conclusion that the fastest barrier with regards to the synchronization time scales with $O(log_2 P \cdot L)$ in the LogP model and also

on real world systems. This scaling can be very time demanding if the barrier operation is frequently used because typical network latencies L vary from $4\mu s$ up to several $100\mu s$. We want to show that a very easy and cheap additional barrier "network" can lead to much higher barrier performance in this context.

The remaining paper is structured as follows. The next subsection summarizes the related work to optimize barrier performance. Section 2 gives a short overview of the current proof of concept implementation. Some performance evaluation data for microbenchmarks as well as an application benchmark is given in section 3, followed by a conclusive summary and an outlook to future research directions.

1.1 Related Work

Enhancing barrier performance has a long history. Several attempts to optimize the synchronization on top of point-to-point messages have been taken and are described in [HMMR04]. The Idea to optimize the barrier by leveraging a special barrier network is also not new in the area of supercomputing. Large systems, as the earth simulator [HYK03], the Blue Gene/L [GBC+05] or differenc Cray supercomputers incorporate also special hardware support for synchronization. A similar low-cost approach has been taken at the Purdue University and is described in [DCMM95]. However, our design is much simpler and applicable to each MPI application without even recompiling it.

2 Design

The design of our hardware barrier is as simple as possible. The current proof of concept hardware uses the parallel port to connect the PC to a central controlling logic. This logic has been implemented in an Altera UP1 FPGA (see figure 1). Each computer uses exactly

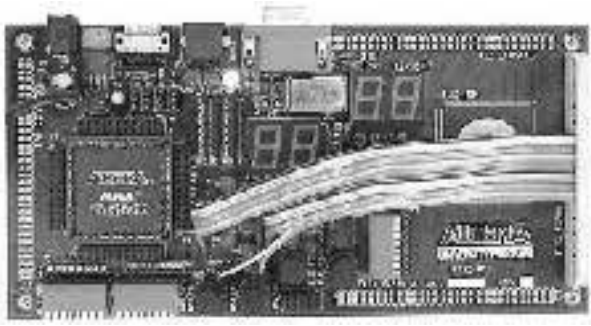

Figure 1: The Altera FPGA

three cables to connect to the FPGA: in, out and ground. A schematic parallel port pinout

is shown in figure 2. We used pin 14 (CONTROL[1]) for our incoming path and pin 2 (DATA[0]) for our outgoing path. Using this scheme, we could offer up to 5 parallel barriers at any time (limited by the number of incoming lines). The central barrier logic

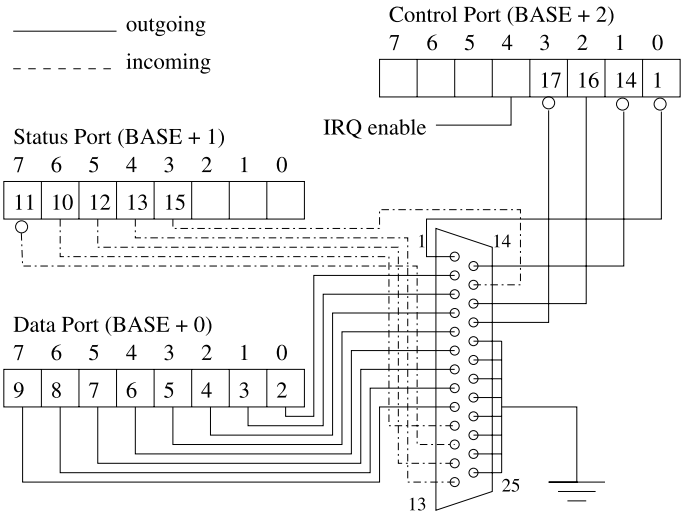

Figure 2: Parallel Port Pin-out

just implements a finite state machine with two states, shown in figure 3. This two state machine toggles the outgoing bit, when all incoming bits are set inversely to the current state. This enables us to use a very simple synchronization scheme which only needs one read and one write operation to the parallel port to synchronize (in the best case, if all nodes reach their barrier simultaneously). The scheme which has to be performed for each barrier call is described in the following:

1. read status (in)
2. toggle status
3. write new status (out)
4. read status (in) until toggled

The first read can be done during the initialization phase and the status can be stored between the barrier calls. So that the scheme is reduced to toggle - write - read.

2.1 Implementation

The support for the hardware barrier has been implemented into the Open MPI [GFB+04] framework as described in [HSM+05]. We shortly describe the methodology in the fol-

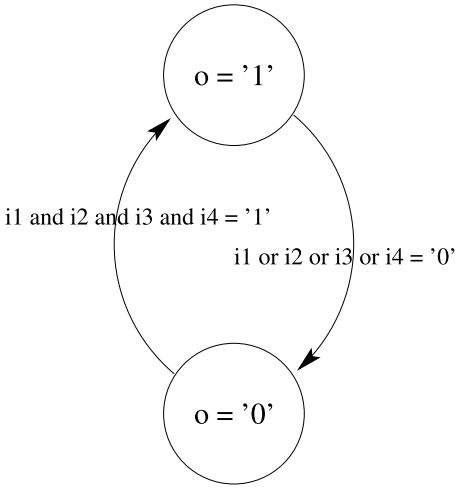

Figure 3: The implemented Finite State Machine

lowing. The hardware barrier was implemented as a collective module in Open MPI. The init phase is used to read the initial status of the hardware and to store this status associated with the communicator. Currently, only MPI_COMM_WORLD is supported by our implementation because we have only two wires and the module simply refuses to run for all other communicators (the framework falls back to another barrier implementation in this case). The barrier call itself is performed with the toggle - write - read schema described above. Another limitation of our current proof of concept design is that the process has to run with root privileges to communicate with the hardware barrier with inb and outb (this could be avoided with a simple kernel patch).

A small code example which shows how the communication with the barrier hardware is implemented is shown in Listing 1

3 Performance Evaluation

The performance of the hardware barrier can be modeled with the following parameters:

- o_w CPU overhead to write to the parallel port
- o_r CPU overhead to read from the parallel port
- $o_p(P)$ processing overhead of a state change
- P number of participating processors

```
#include <stdio.h>
#include <unistd.h>
#include <asm/io.h>

#define BASEPORT 0x378

int main()
{
  /* Get access to the ports - only as root! */
  if (ioperm(BASEPORT, 3, 1)) {perror("ioperm"); exit(1);}

  /* Set the data signals (D0-7) of the port to all low (0) */
  outb(0, BASEPORT);

  /* Read from the status port (BASE+1) and print the result */
  printf("status: %d\n", inb(BASEPORT + 1));
}
```

Listing 1: Accessing the Parallel Port in C

The minimal barrier latency of our toggle - write - read schema (without process skew) can be predicted with:

$$t_b = o_w + o_p(P) + o_r$$

One write is performed to indicate that the barrier is reached and one read (minimum) is performed to test if all nodes reached their barrier.

3.1 Parameter Benchmark

A benchmark of the single parameters returns the following values on our compute cluster consisting of 4 2.4GHz Xeon machines ($1 \leq P \leq 4$):

$$\begin{aligned} o_w &= 1.2\mu s \\ o_r &= 1.2\mu s \\ o_p(P) &= P \cdot 10ns \end{aligned}$$

Thus, the running time can be predicted for our cluster with 4 nodes as:

$$t_b = 2 \cdot 1.2\mu s + 4 \cdot 0.01\mu s = 2.44\mu s$$

This mechanism is extremely scalable because the overall running time is nearly not changed even if the o_p parameter increases linearly.

3.2 Scalability

At first view, the scalability of this approach is limited by the I/O ports of the used FPGA. But one could imagine to increase the number of connected processors with a tree based FPGA scheme (each FPGA acts as a client to the upper layer and sets its barrier reached state, if and only if all its clients have set it). This scheme would scale logarithmically ($O(log_2 P)$) with a very low multiplier (basically o_w or o_r).

3.3 MPI Microbenchmark

We benchmark the MPI_Barrier() latency to evaluate the hardware in a realistic environment. We achieve a barrier latency of $2.57\mu s$ for all four nodes with the Pallas Microbenchmark [Pal00]. This shows that the Open MPI framework is highly efficient and adds only $0.13\mu s$ to the MPI_Barrier() latency.

3.4 MPI Application Benchmark

We use the application Abinit [GBC+02] to benchmark our hardware barrier under real-world conditions with a real-world application. Abinit calculates the ground-state energy of quantum mechanical systems by optimizing the electron-wavefunctions at different so called k-points with an iterative scheme. It calls MPI_Barrier() between each iteration and for the collection of data at the end. The k-points can be calculated independently and communication is only needed at the end of each iteration and at the end of the calculation itself to gather the results. Thus, the k-point parallelization in Abinit is highly parallel and the MPI overhead of the application is only 8%. About 65% of this overhead is caused by MPI_Barrier() calls. A single application run on our Dual-Xeon cluster needs 4:34 min with the standard Open MPI implementation utilizing the software barrier. The usage of the hardware barrier reduced the running time to 4:27 min. The measurement has been repeated 10 times and the variance has been less than 1 second. Due to the massively parallel nature of the application, the improvement is only 2.55%. But if we compare the MPI overhead, we can see that it decreases by 31.77% and the barrier latency is effectively halved.

4 Conclusions and Future Work

We demonstrate an easy and cheap implementation for adding hardware barrier support to commodity clusters. The hardware barrier, including all overheads, needs only $2.5\mu s + 0.01 \cdot P\mu s$ to perform the MPI_Barrier() operation with P nodes in our testing environment. Our Application benchmark shows that the MPI overhead can be reduced significantly in the case of frequent barrier usage. The prototypic nature of our hardware barrier leaves much space for enhancements. The necessity to run as root could be removed with a small kernel patch. Also the number of supported barriers could be increased easily to 5. With a more complex addressing scheme, which increases the latency, up to 2^{11} if 11 of 12 outgoing lines would be used as address vector (the last one remains as status line). Thus, the barrier could be enabled to support 2^{11} different communicators. An implementation in the operating system would also be possible (e.g. as /dev/barrier) and would enable more applications to use it.

References

[CKP+93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[DCMM95] H. G. Dietz, T. M. Chung, T. I. Mattox, and T. Muhammad. Purdue's Adapter for Parallel Execution and Rapid Synchronization: The TTL PAPERS Design. *Technical Report, Purdue University School of Electrical Engineering*, 1995.

[GBC+02] X. Gonze, J.-M. Beuken, R. Caracas, F. Detraux, M. Fuchs, G.-M. Rignanese, L. Sindic, M. Verstraete, G. Zerah, F. Jollet, M. Torrent, A. Roy, M. Mikami, Ph. Ghosez, J.-Y. Raty, and D.C. Allan. First-principles computation of material properties : the ABINIT software project. *Computational Materials Science 25, 478-492*, 2002.

[GBC+05] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, M. E. Giampapa P. Coteus, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49(2):195–213, 2005.

[GFB+04] 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.

[HCM+05] Torsten Hoefler, Lavinio Cerquetti, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. A practical Approach to the Rating of Barrier Algorithms using the LogP Model and Open MPI. In *Proceedings of the 2005 International Conference on Parallel Processing Workshops*, pages 562–569, June 2005.

[HFM88] Debra Hengsen, Raphael Finkel, and Udi Manber. Two Algorithms for Barrier Synchronization. *Int. J. Parallel Program.*, 17(1):1–17, 1988.

[HMMR04] Torsten Hoefler, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. A Survey of Barrier Algorithms for Coarse Grained Supercomputers. *Chemnitzer Informatik Berichte - CSR-04-03*, 2004. url: http://archiv.tu-chemnitz.de/pub/2005/0074/data/CSR-04-03.pdf.

[HSM+05] Torsten Hoefler, Jeffrey M. Squyres, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. Implementing a Hardware-based Barrier in Open MPI. In *Proceedings of 2005 KiCC Workshop, Chemnitzer Informatik Berichte*, pages –, November 2005.

[HYK03] Shinichi Habata, Mitsuo Yokokawa, and Shigemune Kitawaki. The Earth Simulator System. *NEC RESEARCH&DEVELOPMENT, Special Issue on High Performance Computing*, 44(1):21–27, 2003.

[Pal00] Pallas GmbH. Pallas MPI Benchmarks - PMB, Part MPI-1. Technical report, Pallas GmbH, 2000.