A practical Approach to the Rating of Barrier Algorithms using the LogP Model and Open MPI*

Torsten Hoefler Lavinio Cerquetti Torsten Mehlan Frank Mietke
Wolfgang Rehm
{htor, lce, tome, mief, rehm}@cs.tu-chemnitz.de
Chemnitz University of Technology
Chair of Computer Architecture

Abstract

Large–scale parallel applications performing global synchronization may spend a significant amount of execution time waiting for the completion of a barrier operation. Consequently, numerous research works have focused on reducing the communication costs of synchronization primitives. However, so far there has been no exhaustive comparison of barrier algorithms. This paper will investigate significant representatives of this family of algorithms and evaluate their diverging characteristics, with the purpose of assessing their properties within the context of a specific scenario. The first part of this work will introduce four run time complexity classes, to which all barrier algorithms are known to belong. Then, the LogP model will be used to analyze the behavior and predict the running time of a representative algorithm of each class. As these performance predictions will be scrutinized with the help of measurements conducted on original implementations based on the Open MPI framework, this work will show how to leverage the flexible component architecture of this new MPI implementation, which has proved to be an ideal research tool.

1 Introduction

In order to choose a barrier algorithm for implementing or enhancing interprocessor communication in distributed shared memory systems, different aspects have to be considered. All available algorithms vary in some regard, such as network usage and congestion or memory access patterns. The LogP model is often used to simplify the modeling of such algorithms, although its structural assumptions may decrease the accuracy of the prediction. In this paper, a theoretical analysis for several barrier algorithms, each representing a complexity class will be verified on the basis of actual test results. A full textual, graphical and pseudocode description of all barrier algorithms mentioned in this paper can be found in [10]. This may help to understand the split-up into different complexity classes and the LogP running time models for each of the algorithms. The main goal of this paper is to analyze the performance of different barrier algorithms on a central switch based architecture theoretically with the LogP model and practically by benchmarking. Basing on this work, one should be able to select the best algorithm for implementing barrier functionality on similar architectures.
The LogP model is briefly described in section 2. Implementation details for collective components in Open MPI are described in section 3, followed by the benchmark results for a set of algorithms, each acting as a representative of one complexity group, in section 4. Additionally, all results are compared to the predictions of the LogP model. Section 5 weights all algorithms against the native Open MPI Barrier and draws a conclusion towards choosing an algorithm for concrete implementation.

*This work was done in the context of the research project 7455/1180 and funded within the framework for technology promotion by means of the European Fund for Regional Development (EFRE) 2000-2006 as well as by means of the SMWA Saxony ministries.

2 The Model

The widely used LogP model proposed by Culler et al. in 1993 [2] is used as a base for modeling and evaluating the different algorithms.

2.1 Model Description

The LogP model reflects different aspects of coarse grained machines formed by a collection of complete computers, each consisting of one or more processors, cache, main memory and a network interconnect. A main assumption is that the computing power (bandwidth) is much higher than

the network communication bandwidth. The following four parameters compose the LogP model:

- L - communication delay (**upper** bound on the latency for NIC-to-NIC messages from one processor to another)
- o - communication overhead (time that a processor is engaged in transmission or reception of a single message)
- g - gap (indirect communication bandwidth, minimum interval between consecutive messages, $bandwidth \sim \frac{1}{g}$)
- P - number of processors

The parameters of the LogP model can be divided into two tiers, the CPU layer and the network layer. The o-parameter can be further split in one parameter on the receiver side (o_r) and one on the sender side (o_s). A visualization of the role of the different parameters for a given network (e.g. Ethernet) is depicted in Figure 1.

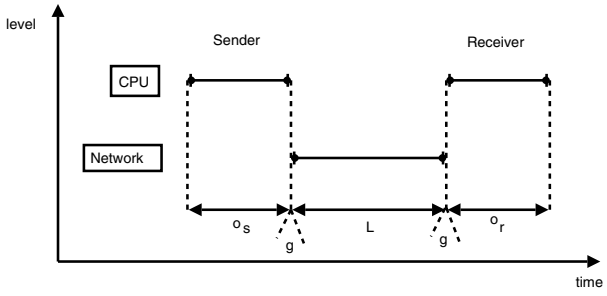

Figure 1. Visualization of the LogP parameters

An additional study [3] describes options for assessing the network parameters of actual supercomputers. By using such approximations to parametrize the LogP model it is possible to estimate the real running time of the mentioned barrier algorithms on real systems.

2.2 Further Assumptions

All nodes are connected through an interconnect network with the following simplifying characteristics:

- full bisectional bandwidth
- full duplex operation (parallel send/receive)
- the forwarding rate is unlimited
- the latency (L from LogP model) is constant above all messages
- the overhead (o) for processing the TCP/IP protocol is bigger than the gap (g) and the latency (L) of the interconnect
- the overhead (o) is constant for all messages (and to simplify $o = o_r = o_s$)

The communication characteristics are defined as follows. The time to send and receive a single message[1] can be approximated to $o_s + L + o_r$, and the time to send n messages[2] can be estimated as $o_s + (n-1)max\{o_s, g\}$[3]. The time to receive n messages (relative to the first packet sent on the sender side) can be modeled as $o_s + L + o_r + (n-1)max\{o_s, g\}$.
Additionally, some constructs show up frequently and are defined as follows:

$$
\begin{aligned}
f_r &= max\{o_r, g\} && (1)\\
f_s &= max\{o_s, g\} && (2)\\
t_r &= max\{f_r, o_s + L + o_r\} \\
&= max\{max\{g, o_r\}, o_s + L + o_r\} \\
&= max\{g, o_s + L + o_r\} && (3)\\
t_s &= max\{g, o_s + L + o_r\} && (4)
\end{aligned}
$$

With the aforementioned assumptions follows

$$
\begin{aligned}
f_r &= f_s = o && (5)\\
t_r &= t_s = 2o + L && (6)
\end{aligned}
$$

Throughout this paper, we will distinguish between t_r and t_s in order to emphasize the semantic properties of the algorithms being analyzed.

3 Implementation with Open MPI

Open MPI, presented in [5], has been chosen as the hosting framework for the barrier algorithms due to its open and extensible nature, and in view of the easiness with which new collective algorithms can be integrated and tested. The general architecture of Open MPI will be briefly described in the following. However, Open MPI is presently undergoing a heavy development phase, and some of its layers have been recently redesigned. As it is still unclear whether new structural changes are about to happen, this section will refer to the current Open MPI pre-release state.

[1] $1:1$ communication
[2] $1:n$ communication with enqueuing
[3] the time o_s and g can run in parallel

3.1 Component Framework

The architecture of Open MPI described in [5] and [13] has changed slightly and can be currently described as consisting of three distinct software tiers:

1. **MPI** - MPI Layer
2. **RTE** - Run Time Environment
3. **MCA** - Modular Component Architecture

The MPI Layer is the adaption layer integrating the MPI standard into the underlying functionality (mainly the RTE and the MCA). The RTE layer provides services at run time (e.g. process start up or output forwarding). As these layers do not have to be modified in order to incorporate new collective algorithms, they will not be investigated further.
The MCA layer is a component framework called Modular Component Architecture (formerly MPI Component Architecture). It manages the layers below by providing several services (e.g. finding components and processing user parameters). Each major functional area has an associated component framework which manages multiple modules performing related or identical tasks. Each component is clearly defined by an interface and offers functional services to the upper tier.
The framework with all its layers, example component frameworks (components A and Z) and managed modules are shown in figure 2.

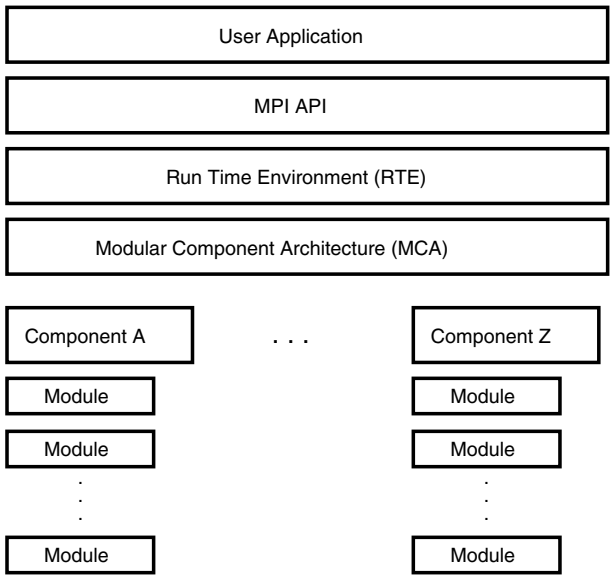

Figure 2. Open MPI Architecture

The next listing shows a number of frameworks already implemented in Open MPI. However, the architecture is open enough to add arbitrary functionality with new frameworks.

- **PTL** - the Point-to-point Transport Layer consists of network specific modules responsible for low-level data transfer. It can be seen as a kind of device driver.
- **PML** - the Point-to-point Management Layer provides several transport services for the MPI Layer (e.g. segmentation and reassembly, striping or reliability).
- **COLL** - the Collective framework provides modules for collective operations.
- **TOPO** - the Topology framework offers processes running within an MPI job a facility which allows the MPI library components to perform locality-based optimizations (e.g. in grid environments).

As this paper will focus on the concerns of the COLL framework, the remaining layers will not be considered further. In order to understand the structure of a single COLL component, the general lifetime of a component has to be described.

3.2 A Components Life-cycle

As described in [13], a component runs through five stages during its existence within the MCA: selection, initialization, checkpoint/restart, normal operation and finalization. Figure 3 shows the order in which these stages are traversed. Each communication is directly associated with a single COLL module, although several of them are allowed to share their source code. In other words, only one instance of a COLL component delivering a specific functionality can be active at any time for a specific communicator, and the state of each COLL module is relative to the hosting communicator.

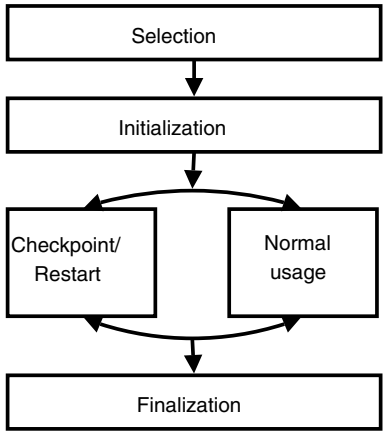

Figure 3. A Components Lifecycle

The ”selection” is done during the creation of a new communicator (including `MPI_COMM_WORLD` and

MPI_COMM_SELF), typically triggered inside the MPI API functions MPI_INIT, MPI_COMM_CREATE, MPI_COMM_DUP or MPI_COMM_SPLIT. Due to the fact that no special hardware support is needed, the component does not need to check for it and returns the configured priority (swbarr_priority), or 20 by default.
The winning component enters the "initialization" phase and the COLL framework calls the mca_coll_<name>_module_init function. The component initializes all data structures and calculates the communication partners for each algorithm and round in advance to speed the critical path up. The used algorithm can be selected as mca-parameter (swbarr_selection). If no algorithm is selected, the default will be 0, which means that each algorithm is benchmarked for this communicator and the fastest one is chosen automatically. Several additional mca-parameters are read during this phase and saved for the communicator (e.g. the n-parameter for the Combining Tree).
The "checkpoint/restart" stage has to take care of messages which are currently on the fly and has to drain all queues. The COLL module used in this paper does not implement directly this functionality, but rather delegates it to the underlying PTL components.
"Normal usage" is the state in which the requested collective operations (e.g. MPI_Barrier()) are actually performed. Previously stored data may be extracted from the communicator on as-needed basis, and the selected functionalities are activated by way of the corresponding function pointer provided to the COLL in the initialization phase.
The finalization step requests the module to clean up all used data structures and drain the network in order to unload cleanly. The function mca_coll_<name>_module_finalize is called to trigger the cleanup.

4 Benchmark Results

This section will present benchmark results for the representative algorithms of each complexity class. Four complexity domains for barrier algorithms can be identified on the basis of the LogP model:

1. $O(P) \Rightarrow$ Central Counter [4, 6]
2. $O(n \cdot log_n P) \Rightarrow$ Combining Tree [15], f-way Tournament [7] and MCS [11]
3. $O(log_2 P)$ with broadcast $\Rightarrow$ Tournament [9] and BST [14]
4. $O(log_2 P)$ without broadcast $\Rightarrow$ Butterfly [1], Pairwise Exchange [8] and Dissemination [9]

All algorithms have been implemented in a new COLL component within the Open MPI framework, as described in section 3. The dynamic algorithm selection and the input of configuration values (mainly the group size n of the Combining Tree Barrier) have been realized by utilizing the mca_parameter functions, which can be used to parametrize the module during runtime.
The resulting code was executed on our local cluster, consisting of 528 Pentium III 800 MHz nodes interconnected with an Extreme Black Diamond 6x96-Port Fast Ethernet switch. This switch satisfies nearly all the requirements stated in section 2.2.
The results achieved using the Pallas Micro-Benchmark [12] and their analysis according to the LogP model will be shown in the following sections of this paper.

4.1 Central Counter

The central counter is already implemented in the Open MPI framework, as the runtime for small sets of processors is extremely low, even though the scaling with processor count is still suboptimal. Currently, the Open MPI framework defines a threshold processor number (as an MCA parameter) used to switch the barrier logic between the central counter and another logarithmic implementation. This feature was disabled during the tests in order to ensure the utilization of the former algorithm. The results are shown in figure 4. The algorithm runtime prediction in the LogP

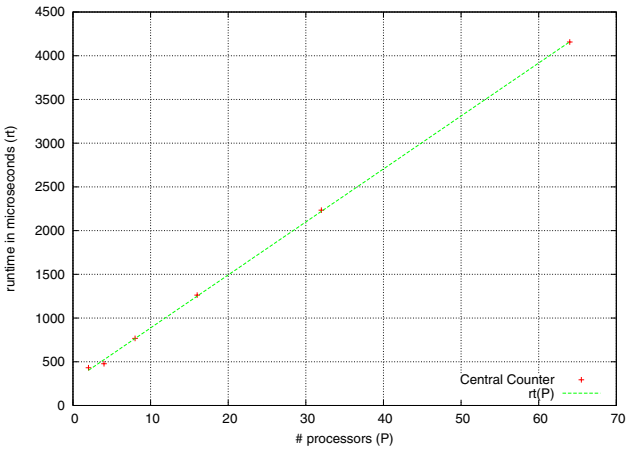

Figure 4. Central Counter

model is shown in figure 5. The left part denotes phase 1, where each processor $P > 0$ sends a packet to processor $P = 0$, that it reached the barrier. The abscissa presents the time (t) and this illustration assumes that all processors reach their barrier at $t = 0$. Even if all processors send simultaneously and the latency is equal for all of them,

the packets are serialized at the receiver, because a received packet "blocks" the network interface for the time f_r. Phase 2 starts immediately after all packets have been received. The same blocking appears during the send operation, because every sent packet "blocks" the sender for the time f_s. All following LogP figures can be read in the same way. Phase 1 is finished after

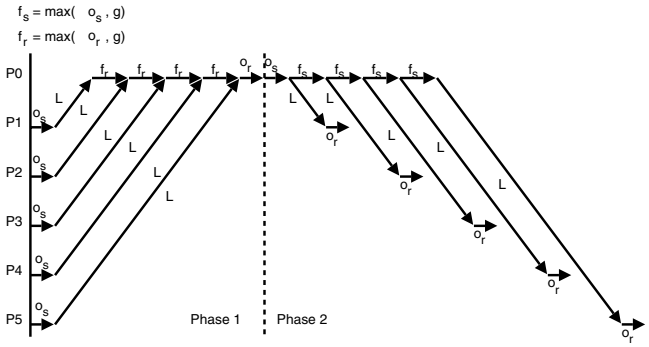

Figure 5. LogP for Central Counter

$$rt_{phase1} = o_s + L + (P-2)f_r + o_r$$

Phase 2 starts at $T = rt_{Phase1}$ and the runtime until the last node is notified can be predicted as

$$rt_{phase2} = o_s + (P-2)f_s + L + o_r$$

whereby Node 0 finished the barrier after

$$\begin{aligned} rt_{node0} &= rt_{phase1} + o_s + (P-2)f_s \\ &= 2o_s + o_r + L + (P-2)f_r + (P-2)f_s \end{aligned}$$

Node $i \in \{1, \cdots, P\}$ finishes after

$$\begin{aligned} rt_{nodei} &= rt_{phase1} + o_s + L + (i-1)f_s + o_r \\ &= 2(o_s + L + o_r) + (P-2)f_r + (i-1)f_s \end{aligned}$$

The last Node (P) and the whole barrier finishes after

$$rt = 2(o_s + L + o_r) + (P-2)f_r + (P-2)f_s$$

Impact of the Pallas Benchmark Loop
The Pallas Benchmark makes use of a loop, which by default is executed 1000 times ($b = 0; b < 1000; b++$), to measure the running time of the barrier operation. This loop could have an impact on our results, due to the fact that some nodes may enter a new barrier ($b+1$) before all nodes have completed the previous one (b). This is namely the case in the central counter, as P1 finishes the barrier after

$$rt = 2(o_s + L + o_r) + (P-2)f_r$$

and sends its packet for the new barrier $b+1$ to P0, which is still sending packets related to the previous barrier b. The first packet arrives and is enqueued by the MPI library, as no matching receive was posted yet. P0 posts the first receive after it completes barrier b. The operating system has already processed the message and the MPI layer has stored it in a buffer, so f_r has been paid already (by delaying barrier b) for the first messages, when P0 enters barrier $b+1$. This adds a constant overhead to barrier b in each round (processing o_r for messages of barrier $b+1$), and it is easy to see that the asymptotic behavior is not changed.
rt can be simplified under the assumptions taken in 2.2 and a large processor count P:

$$\begin{aligned} rt &= 2(o_s + L + o_r) + (P-2)f_r + (P-2)f_s \\ o &= o_r = o_s \\ f_r &= f_s = o \; (o > g) \\ rt &\approx 2(2o + L) + 2o(P-2) \\ &\approx 2(L + Po) \end{aligned}$$

Thus, we expect the runtime (rt) to behave like a linear function.

4.2 Combining Tree

The Combining Tree algorithm is used to represent a $O(n \cdot log_n P)$ algorithm. According to the LogP model, the runtime of phase 1 can be assessed as shown in figure 6. The runtime can be predicted with

$$rt_{phase1} = (o_s + L + f_r(n-2) + o_r) \cdot \lceil log_n P \rceil$$

as an upper bound. $\forall P \neq n^x \; (x \in \mathbb{N})$ the running time for phase 1 is slightly smaller.
Phase 2, denoted as $t_{bc}(P-1)$, leverages the binomial tree algorithm in Open MPI and is analyzed in figure 7. The runtime can be predicted as

$$\begin{aligned} t_{bc}(P-1) &= o_s + (\lceil log_2 P \rceil - 1) \\ &\quad \cdot max\{f_s, o_s + L + o_r\} + L + o_r \\ &= o_s + (\lceil log_2 P \rceil - 1)t_s + L + o_r \end{aligned}$$

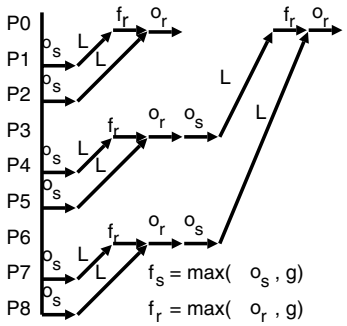

Figure 6. LogP for Combining Tree ($n = 3$)

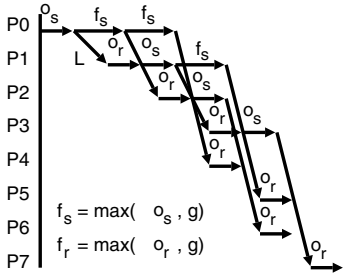

Figure 7. LogP for the Broadcast Algorithm

Thus, the whole runtime rt can be seen as

$$\begin{aligned} rt &= (o_s + L + f_r(n-2) + o_r) \cdot \lceil log_n P \rceil + o_s \\ &\quad + (\lceil log_2 P \rceil - 1)t_s + L + o_r \end{aligned}$$

rt can be simplified under the assumptions taken in 2.2 and a large processor count P:

$$\begin{aligned} rt &\approx (L + no) \cdot \lceil log_n P \rceil + \lceil log_2 P \rceil \cdot (2o + L) \\ &\approx log_2 P \cdot (2o + L) \end{aligned}$$

The optimization of rt with the fitted values for the parameters o_s, o_r and L states the existence of a global minimum for $n = 4$. This result is endorsed by the benchmarks shown in figure 8 and confirms the predictive power of the LogP model, as well as the correctness of the parameter assessment and the validity of our algorithmic analysis.

4.3 Tournament Barrier

The Tournament Barrier is chosen to represent the class of $O(log_2 P)$ algorithms which perform the last step of notifying all other nodes, typically by broadcasting to them.

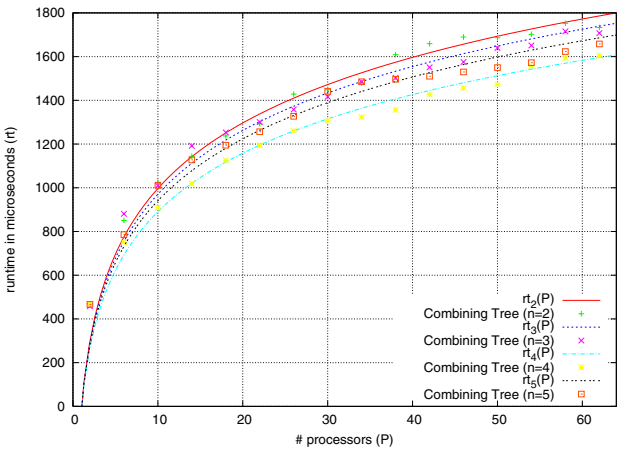

Figure 8. Measured rt Values

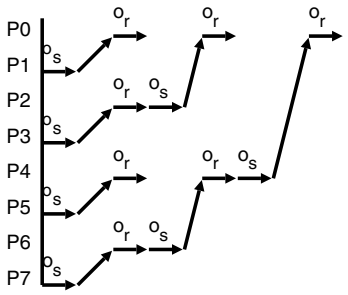

Figure 9. LogP for the Tournament Barrier

The LogP prediction which is depicted in figure 9 can be described as (for $\forall P = 2^j \ (j \in \mathbb{N})$, for all other P, the running time is slightly lower):

$$rt = t_r \cdot \lceil log_2 P \rceil + t_{bc}(P-1)$$

The Binomial Tree is used again for broadcasting at the end:

$$t_{bc}(P-1) = o_s + (\lceil log_2 P \rceil - 1)t_r + L + o_r$$

Assuming the usual simplifications

$$\begin{aligned} rt &= t_r \cdot \lceil log_2 P \rceil + o_s + (\lceil log_2 P \rceil - 1)t_s \\ &\quad + L + o_r \\ &\approx (t_r + t_s) \cdot \lceil log_2 P \rceil + o_s + L + o_r \\ &\approx 2(2o + L)\lceil log_2 P \rceil \end{aligned}$$

The matched functions and measured values are shown in figure 10.

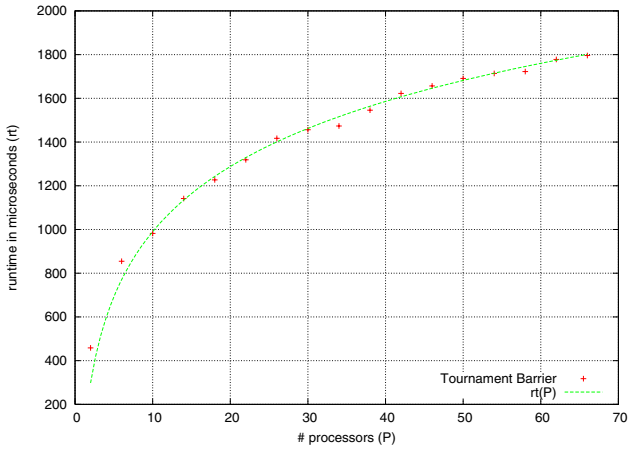

Figure 10. Tournament Barrier

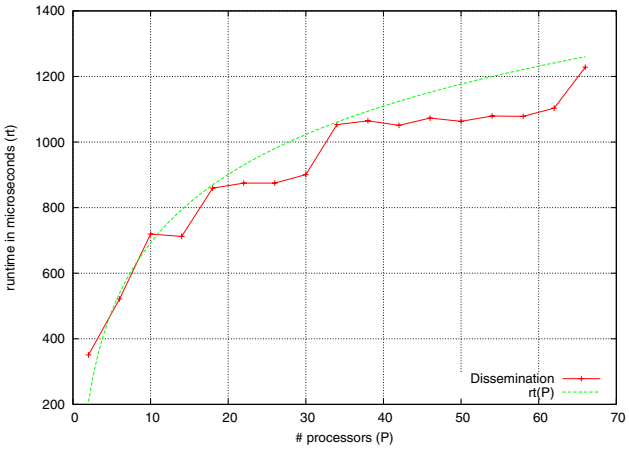

Figure 12. Dissemination Barrier

4.4 Dissemination Barrier

The Dissemination Barrier, belonging to the complexity class $O(log_2 P)$ of algorithms which do not need to go through a final broadcast phase, was previously proven to deliver an optimal solution to the barrier problem [10]. Thus, we reasonably expect this algorithm to provide the best results. The LogP modeling is shown in figure 11 and the runtime can be predicted with

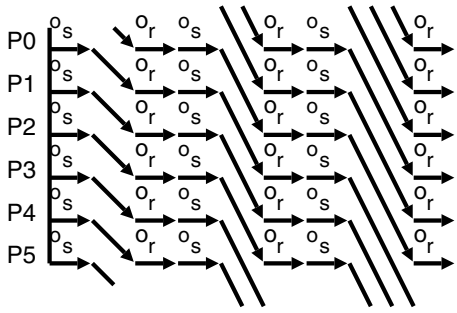

Figure 11. LogP for the Dissemination Barrier

$$rt \quad = \quad max\{t_r, t_s\} \cdot \lceil log_2 P \rceil$$

With the usual simplifications, the runtime behaves asymptotically as follows

$$rt \quad = \quad (2o + L) \cdot \lceil log_2 P \rceil$$

The benchmark results and a fitted function for the upper bound are shown in figure 12. The running time increases in steps, with each step bound to a successive power of two in the process count.

Table 1. Big Numbers of Processors

Algorithm	128 nodes	256 nodes
Central Counter	$4594.50\mu s$	$4909.67\mu s$
Combining Tree	$4009.79\mu s$	$4343.63\mu s$
Tournament	$3642.54\mu s$	$4378.77\mu s$
Dissemination	$1904.57\mu s$	$1977.12\mu s$
Open MPI	$3559.88\mu s$	$4226.88\mu s$

4.5 Comparison of the Different Algorithms

We have shown in sections 4.1 to 4.4 that the LogP model is able to provide very accurate prediction on asymptotic behavior of barrier algorithms for systems complying with the model assumptions [4]. As expected, the Dissemination Algorithm delivers the best results and confirms itself as the optimal solution to the barrier problem on LogP compliant systems. All measured data up to 64 nodes is shown in figure 13. Some results of the benchmarks conducted on 128 and 256 nodes are shown in table 1. Open MPI means the native implementation which switches between a Central Counter and the Binomial Tree, but due to the big number of processors, the latter is used.

5 Conclusion

This work has shown that on distributed memory systems which comply with the assumptions of the LogP model the Dissemination Algorithm represents an optimal solution to

[4] the asymptotic standard error has been less than 5% for every fitting

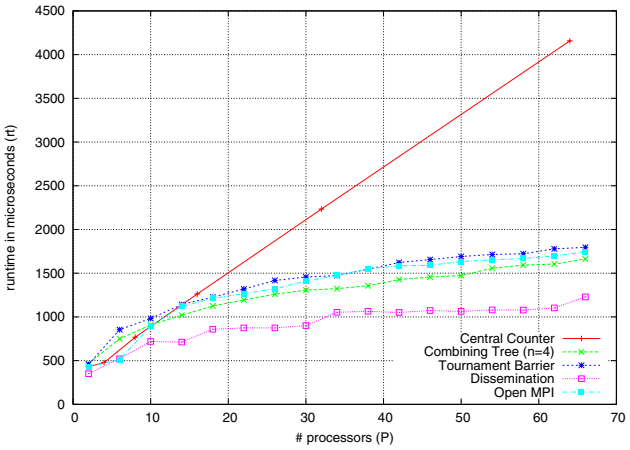

Figure 13. Comparison of all Barrier Algorithms

the barrier problem. However, this result can not be generalized to all system classes and configurations, e.g. oversubscribed switch-based systems with no guaranteed full bisection bandwidth. In such scenarios, an adaptive behavior of the communication layer could be beneficial. According to the focus of the present paper, we implemented an Open MPI component which dynamically conducts a benchmark of barrier algorithms in the initialization phase and select the most effective one for each communicator. Moreover, the basic barrier algorithm of Open MPI could be replaced with the Dissemination Algorithm, as the compliance with the LogP model can be reasonably assumed for modern supercomputers. Generally speaking, all barrier primitives should make use of the fastest algorithm available under the specific architecture. The work shown in this paper and the pseudocode from [10] can be used to determine an optimal barrier algorithm for central switch based interconnect.

References

[1] E. D. Brooks. The Butterfly Barrier. *International Journal of Parallel Programming*, 15(4):295–307, 1986.

[2] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[3] D. Culler, L. T. Liu, R. P. Martin, and C. Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *IEEE Micro*, February 1996.

[4] E. Freudenthal and A. Gottlieb. Process Coordination with Fetch-and-Increment. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 260–268. ACM Press, 1991.

[5] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. 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] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. *SIGARCH Comput. Archit. News*, 17(2):64–75, 1989.

[7] D. Grunwald and S. Vajracharya. Efficient Barriers for Distributed Shared Memory Computers. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 604–608. IEEE Computer Society, 1994.

[8] R. Gupta, V. Tipparaju, J. Nieplocha, and D. Panda. Efficient Barrier using Remote Memory Operations on VIA-Based Clusters. In *2002 IEEE International Conference on Cluster Computing (CLUSTER 2002)*, page 83. IEEE Computer Society, 2002.

[9] D. Hengsen, R. Finkel, and U. Manber. Two Algorithms for Barrier Synchronization. *Int. J. Parallel Program.*, 17(1):1–17, 1988.

[10] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm. A Survey of Barrier Algorithms in the Context of the LogP Model and Proof of Optimality. *Chemnitzer Informatik Berichte - CSR-04-03*, 2004.

[11] J. Mellor-Crummey and M. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

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

[13] J. M. Squyres and A. 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.

[14] N.-F. Tzeng and A. Kongmunvattana. Distributed Shared Memory Systems with Improved Barrier Synchronization and Data Transfer. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 148–155. ACM Press, 1997.

[15] P. Yew, N. Tzeng, and D. Lawrie. Distributing Hot Spot Addressing in Large Scale Multiprocessors. *IEEE Trans. Comput.*, 36(4):388–395, 1987.