Parallel Computing 46 (2015) 14–31

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

τ-*Lop*: Modeling performance of shared memory MPI

Juan-Antonio Rico-Gallego [a,*], Juan-Carlos Díaz-Martín [b]

[a] *Department of Computer Systems Engineering and Telematics, University of Extremadura, Avd. Universidad s/n, 10003 Cáceres, Spain*
[b] *Department of Computer and Communications Technology, University of Extremadura, Avd. Universidad s/n, 10003 Cáceres, Spain*

ARTICLE INFO

Article history:
Received 17 September 2013
Received in revised form 11 February 2015
Accepted 23 February 2015
Available online 5 March 2015

Keywords:
Formal models
Performance analysis
Parallel algorithms
MPI collectives

ABSTRACT

Formal modeling of the cost of MPI primitives allows a machine independent representation, comparison and performance analysis of their underlying algorithms. Current accepted methods are all the off-springs of *LogP*, conceived to model the cost of inter-node point-to-point messages in networks of single-processor machines. As new supercomputers are built upon cheap commodity boards with a growing number of cores accessing hierarchical memories, intra-node communication becomes progressively more relevant. Techniques for shared memory communication, such as message segmentation and collectives, not based on point-to-point operations, are substantively different from their inter-node counterparts. This paper unveils the reasons for the poor fit of *LogGP* and the most recent models in this domain, $\log_n P$ and $m \log_n P$, and proposes a new model named τ-*Lop*, rooted on them, but addressing the challenge of accurately modeling shared memory MPI communications. Broadcast algorithms of mainstream MPI implementations, *MPICH* and *Open MPI*, are modeled and analyzed.

1. Introduction and goals

Twenty years ago parallel machines were converging to an emerging architecture, clusters of workstations, now known as multicomputers and, more recently, as multi-core clusters. Cost model *LogP* [1,2] addressed the performance of the network with the goal of guiding the design of a parallel algorithm, i.e., organizing the computing loads and the communicating events (point-to-point messages) to optimize its execution time in the cluster. *LogP* characterizes the network performance based on the four parameters that together form its name: network delay (L), the overhead or time invested by the processor in sending or receiving a message (o), the minimum time interval between message transmissions (g) and the number of processors (P) in the cluster. The execution time of a parallel algorithm is now formulated in terms of these parameters with the goal of analyzing the performance of the algorithm in the cluster. *LogGP* [3] extends *LogP* with an additional parameter, G, which captures the network bandwidth for long messages. Several models draw from *LogGP* contribute with a variety of add-ons. *LogGPS* [4] covers aspects of synchronization, providing the *rendezvous* cost, *LogGPH* [5] supports representation for hierarchical architectures and *LoGPC* [6] adds contend time in networks.

Another branch of interest in *LogP/LogGP* [7] is its ability to model the underlying message-passing algorithms that implement the primitives of the MPI standard [8]. MPI modeling is a critical issue, because it allows a formal machine-independent performance comparison and analysis of different implementations of the standard [9–12]. A well known profiling study [13] shows that MPI applications spend more than eighty per cent of communication time in collective operations; hence,

* Corresponding author. Tel.: +34 645269389 (mobile), +34 927257000x51655 (work); fax: +34 927257202.
E-mail addresses: jarico@unex.es (J.-A. Rico-Gallego), juancarl@unex.es (J.-C. Díaz-Martín).

http://dx.doi.org/10.1016/j.parco.2015.02.006
0167-8191/© 2015 Elsevier B.V. All rights reserved.

efficient algorithms for *broadcast* or *reduce* have attracted special interest [7]. To get a feel of collective cost modeling, the well known *binomial tree* broadcasting algorithm is used here. A process named *root* sends data to the rest of the processes in the *communicator*, an isolated group of P processes identified by their *rank* numbers. Fig. 1 shows the algorithm deployment. Process *root* sends a message to the process with rank $root + P/2$. The algorithm continues recursively, with these processes as *roots* of their own trees with $P/2$ processes. It completes in $\lceil \log_2 P \rceil$ stages, the height of the tree; so, the cost of the algorithm is $T = (\alpha + m\beta) \times \lceil \log_2 P \rceil$ under the Hockney [14] model, m being the size of the message, and α the latency and β the inverse of the bandwidth for the specific network.

Current MPI implementations adopt common well established point-to-point algorithms in the inter-node[1] communication, as the *binomial tree* examined above. However, each implementation develops its own techniques in the intra-node domain. Two popular MPI implementations, *MPICH* [15] and *Open MPI* [16], are considered here. *MPICH* builds intra-node collectives upon point-to-point communication library *Nemesis* [17]. *Nemesis* communicates two processes in the same node by lock-free message queues of 64 KB memory blocks called *cells*. Queues and cells are allocated in a memory area mapped by each local pair of processes. *Open MPI* promotes a software architecture based on components (*MCA*, for Modular Component Architecture). Every functionality is paid by a well defined interface known as *framework*. An MCA framework uses the MCA's services to find and load *components* at run time. An MCA component is a standalone collection of codes that can be inserted into the *Open MPI* code base at run-time and/or compile-time. *COLL* is a framework for collectives, including the *SM* (shared memory) component. The key point here is that *Open MPI COLL SM* does not use point-to-point communication to build collectives. Instead, it maps a common memory zone that is written by a given process and simultaneously reads the rest. As a consequence, point-to-point *LogP/LogGP* is not the indicated modeling tool for *Open MPI* on shared memory. Besides, the difficulties of *LogGP* in modeling shared memory performance seem to have been tacitly recognized by the HPC community. In fact, to the best of the authors' knowledge, no report exists on the practical ability of the *LogP* family of algorithms to model the performance of MPI in shared memory. Yet, *LogGP* has been incorporated in the discussions from a formal point of view and its parameters were measured in the studied platforms.

The model $\log_n P$ [18,19] has nothing to do with *LogP*, although they are similar in name. $\log_n P$ was also conceived, just as *LogP*, to model point-to-point communication, but with a new key feature, each individual data transfer that occurs in a message transmission. In the authors' view, the *transfer* is the building block that conveniently suits the purpose of modeling the behavior of MPI algorithms in shared memory. $m\log_n P$ [20] extends $\log_n P$ by distinguishing the variety of communication channels in a system, such as shared memory and network. The locality of a process determines the communication channel used, and so the communication cost. As this paper focuses on shared memory, $m\log_n P$ and $\log_n P$ become equivalent.

The goal of this paper is to propose and explore a new performance model named *τ-Lop*. It is rooted on the concept of transfer, just as $\log_n P$, with the capability of an accurate cost prediction of MPI collectives on shared memory. It will be shown that it is also capable of capturing issues that are specific to shared communication channels, e.g., the formerly mentioned simultaneous access to shared memory by all the processes involved in a collective, or message segmentation. *τ-Lop* allows an expressive, straightforward and hierarchical representation of algorithms, covering the point-to-point approach of *MPICH*, as also the alternative mechanisms adopted by *Open MPI*.

The reminder of this exposition is organized as follows. Section 2 revisits the *LogGP* and $\log_n P$ models and examines their weaknesses in shared memory; Section 3 describes the new model *τ-Lop*; Section 4 explains its application to key algorithms of MPI collectives, besides introducing its potential of analytic inference; Section 5 details the experimental approach used to estimate the parameters of the model and Section 6 concludes.

2. Representative models and their limitations

MPI collective algorithms perform in a hierarchical way on multi-core clusters. In a broadcast, for instance, first, an inter-node broadcast is executed between the representative process in each node, and then a local shared memory broadcast runs inside the node with the representative acting as root [21]. This section shows how current models work on the shared memory stage, besides providing an insight into their drawbacks and mistakes.

Eq. (1) models a typical point-to-point, non-segmented message transmission of size m as a function of the *LogGP* parameters.

$$T^{LogGP} = 2o + L + (m-1)G \tag{1}$$

Segmentation of messages is a technique commonly used by *MPICH* and *Open MPI*. It operates by breaking up the message into smaller chunks known as segments and sending them in sequence. On shared memory, as will be explained later, the segments allow overlapping of the sender copy (to a shared buffer) with the receiver copy (from the shared buffer), accelerating the communication. Fig. 2 represents a point-to-point transmission of a message, split into k segments ($k = 3$) of size S. Eq. (2) provides its cost [7].

$$T^{LogGP} = 2o + L + (S-1)G + (k-1)(g + (S-1)G) \tag{2}$$

[1] The term *intra-node* refers to the communication between two processes by means of shared memory and the term *inter-node* to communication by means of a network.

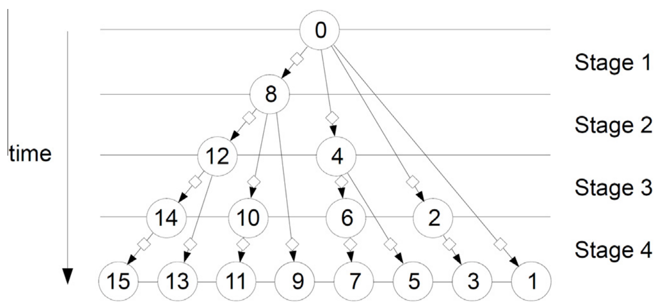


Fig. 1. Broadcasting algorithm as a *binomial tree* for $P = 16$ processes and $\lceil \log_2(P) \rceil = 4$ stages. An arc is a *message transmission* between two processes. Processes are identified by rank, 0 being rank of the root.

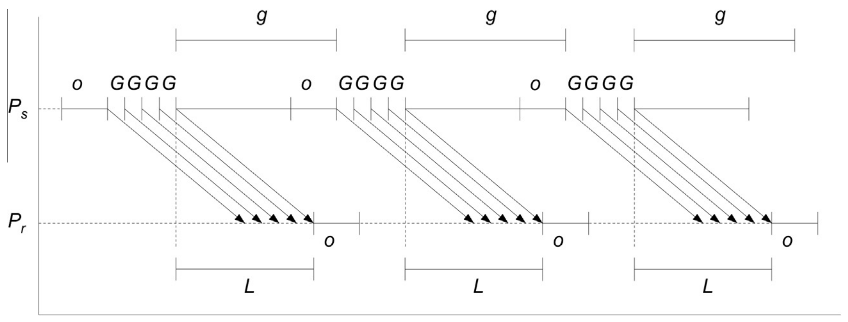


Fig. 2. A point-to-point message represented as the overhead (o) plus the gap per byte (G) and the time to send the next message as the maximum of the latency of the network plus the reception overhead and the gap per message ($max\{L + o, g\}$).

As *LogGP* understands the *binomial tree* algorithm as a mere sequence of parallel point-to-point transmissions, *LogGP* estimates its cost by simply multiplying the cost of a stage with the height of the tree:

$$T_{bcast}^{LogGP} = \lceil \log_2(P) \rceil \times T^{LogGP} \tag{3}$$

As a result, for instance, the cost for $P = 65$ and that for $P = 128$ in a multi-core will be the same, which is far from being a correct prediction.

Model $\log_n P$, on the other hand, assumes that a message transmission takes place between two processes as a sequence of implicit transfers (copies) between source and destination buffers [20]. The cost of a transmission is the sum of the costs of its n individual transfers:

$$T^{\log_n P} = \sum_{i=0}^{n-1} (max\{g_i, o_i\} + l_i) \tag{4}$$

where o (*overhead*) is the length of time the processor is engaged in each transfer for a contiguous message, which is represented by g (*gap*) when contend time is considered, and l (*latency*) is the additional length of processor time when the message is not contiguous in memory, and this leads it to be packed before being sent and unpacked after being received. Though segmentation has a strong impact on MPI intra-node performance, it is not explicitly considered in $\log_n P$, so that segmented and non-segmented transmissions give different values of g and o. Fig. 3 shows an example of a message transmission in shared memory. Here $n = 2$, because the transmission needs just two transfers going through an intermediate buffer. On contiguous messages $l_i = 0,\ \forall i$ and, if the next reasonable simplifications $max\{o_i, g_i\} = o_i$ and $o_i = o\ \forall i$ apply, then (4) becomes

$$T^{\log_n P} = 2o \tag{5}$$

The cost of the *binomial tree* broadcast algorithm in $\log_2 P$ leads to the same expression as the one in *LogGP*:

$$T_{bcast}^{\log_2 P} = \lceil \log_2(P) \rceil \times T^{\log_n P} \tag{6}$$

In our opinion, *LogGP* and $\log_n P$ lack accuracy in shared memory, because they cannot reach adequate level of expressive power for two reasons: concurrent copies and synchronization. As all the cores share the communication channel, the cost of a transfer depends not only on the size of the message, but also on the number of transfers happening simultaneously, an aspect that these models ignore. Concurrent transfers appear, for instance, in the transmissions at the same stage in the

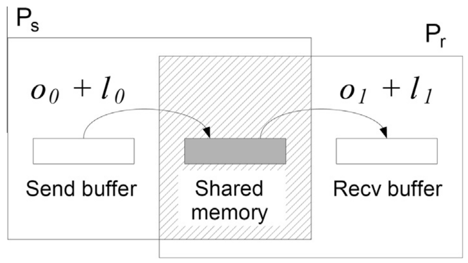


Fig. 3. Concept of *message transmission* in $\log_2 P$. P_s and P_r are the sender and receiver processes respectively. The message transmission occurs in two *transfers* through the shared memory channel. Each transfer has different cost parameters, but equaling them is a reasonable approach in this context.

binomial tree algorithm of Fig. 1, as well as on collective operations that are not built upon transmissions, such as those found in *Open MPI COLL SM*, as will be discussed in Section 3.1. Although we think that the transfer-centric representation of $\log_n P$ is the starting point to model this type of algorithms, a transfer is not a bare copy. The cost of message passing in shared memory includes the additional burden of process synchronization, a parameter which again is not explicit in $\log_n P$. Thus, it is not clear how $\log_n P$ can model operations in which there is no data transfer at all, and hence synchronization time represents the only cost, as in *MPI_Barrier*. As a result, these models present pitfalls in shared memory, as the following example illustrates. The *LogGP* modeling of the well known *Scatter* and *Recursive Doubling Allgather (RDA)* algorithms, which are described later, produces surprisingly the same cost formula [7]:

$$T^{LogGP}_{Scatter} = T^{LogGP}_{RDA} = \log P \times (L + 2o - g) + (P - 1)k(g + (S - 1)G), \tag{7}$$

where $m = k \times S$ is the message size specified in the *MPI_Scatter* and *MPI_Allgather* operations. This prediction is far from being correct. First, *LogGP*, though the number of stages in both algorithms is the same ($\log_2 P$), cannot represent that the whole amount of data moved in *RDA*, $(P - 1) \times m$, exceeds the *scatter* figure of $\log P \times m$; second, the number of processes doing transfers in each stage of *RDA* is higher. Section 4.3 studies in depth the two algorithms under τ-*Lop*. The model confirms *RDA* as a much heavier algorithm. A more expressive representation of collective communication algorithms, and hence a more accurate prediction of its performance in shared memory, motivates the τ-*Lop* approach.

3. Modeling with τ-*Lop*

This section describes the proposed model, whose key challenge is to accurately model the cost of MPI operations, both point-to-point and collective, when deployed on channels that allow concurrent transfers as shared memory. τ-*Lop* can model the features listed below:

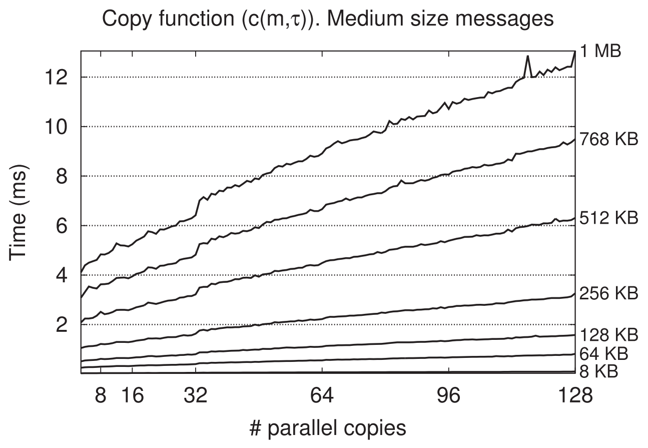
Concurrent transfers: A cost model has to consider the fact that channel bandwidth shrinks when transfers are concurrent. The cost of a copy is represented as $c(m, \tau)$, which is a function of message size (m) and of the number of concurrent copies on the channel (τ). Fig. 4 represents the cost of broadcasting a buffer in core 0 to the rest of the cores of the machine. If all the copies are simultaneous (concurrent), the memory bandwidth gets exhausted. As a result, the cost grows linearly with the number of cores. Current models do not consider this fact.

Collective transfers: A transfer need not be just between a sender and receiver, a restriction of $\log_n P$ and *LogGP* and their derivatives. As will be shown later, *Open MPI COLL SM* implements the collective operation *MPI_Bcast* using shared memory regions that are written by a sender and read simultaneously by several receivers in a collective transfer.

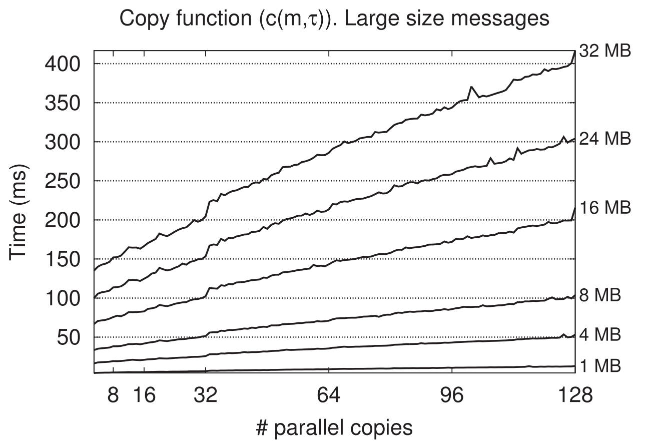
Message segmentation: Transmissions that require an intermediate copy, as those between processes in shared memory, generally use message segmentation techniques. Segmentation demands less intermediate memory because storing the whole message is not needed. In addition, it speeds up the communication progress by overlapping the sending of segments with their reception. However, it comes with the additional cost generated by synchronization of the processes in the interchange of segments, as well as by the effect of concurrent copying by the sender and the receiver. Other techniques do exist for improving message transmission in shared memory. Operating system modules, such as *LiMIC* [22] and *KNEM* [23], provide services of data movement between processes without an intermediate copy, which is supported by τ-*Lop* too.

Costs attributable to protocol: The MPI standard includes a wide range of communication modes that include the synchronous send (*MPI_Ssend*), which enforces sender to wait until receiver arrives. Limitations on buffer space impose MPI libraries to set up communication protocols, such as *rendezvous*, so that sender waits for receiver, who on arrival notifies the sender to proceed. Protocols charge additional cost, mainly at the beginning of the communication, which must be considered in a model.

Computational cost: Collectives such as *MPI_Reduce*, which involve application of an algebraic operation to the message data, add cost to the communication. Its modeling is direct, but beyond the scope of this article.



(a) Regular messages



(b) Large messages

Fig. 4. Cost of a copy as a function of message size (m) and the number of processes concurrently copying (τ) in *Lusitania*, a 128-core machine. All processes copy from the same memory buffer, stressing the communication channel.

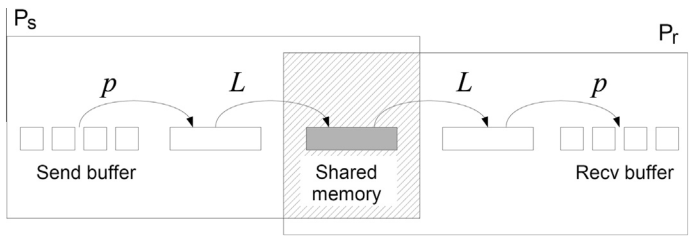


Fig. 5. A *message transmission* in τ-*Lop* is composed by two *transfers* on the shared memory channel.

3.1. The parameters of the model

To represent the foregoing features, a set of six parameters are defined below for τ-*Lop* (see Fig. 5):

L: *Transfer time* (L) is the cost of a transfer and is represented as $L(m,\tau)$. As the transfer may flow concurrently with others (see Fig. 1), its cost depends not only on message size (m), but also on the number of such concurrent transfers (τ). The definition of L, hence, enforces the restriction that the cost of A concurrent transfers will be some value between the cost of a single transfer and that of A consecutive ones, $L(m,1) \leqslant L(m,A) \leqslant A \times L(m,1)$, an expression that is generalized for convenience as follows:

$$L(m,\tau) \leqslant L(m, A \times \tau) \leqslant A \times L(m,\tau) \tag{8}$$

In a transfer, the model considers not only the cost that is attributable to the bare copy ($c(m,\tau)$), but also the cost due to *synchronization management* ($y(\tau)$), defined as the time that the processes involved in the τ transfers invest on the synchronizing resources. Each of the τ transfers involves normally two processes, except in the special case of collective transfers. $y(\tau)$ can represent the time spent polling a flag or using synchronization objects, such as *mutexes* or lock-free queues. The *Transfer time* is hence defined as the sum $L(m,\tau) = c(m,\tau) + y(\tau)$. Though $y(\tau)$ is difficult to measure empirically, it has been observed to dominate the global cost of tiny messages. In particular, as $c(0,\tau) = 0$, the cost to transfer a null message is due to synchronization alone. This definition allows τ-*Lop* to model the cost of operations without data transfer as *MPI_Barrier*, where synchronization is the exclusive cost.

p: Packing cost (p) is the cost due to *pack* and *unpack* of a non-contiguous message. It depends on message size $p(m)$, as well as on the layout of message fragments in memory. It applies only to the first and last transfers of the operation, be it point-to-point or collective. For simplicity, only contiguous messages ($p(m) = 0$) are considered in the following.

o: Overhead (o), despite its name, has nothing to do with the parameter of that name in $\log_n P$. It is defined as the time elapsed since the invoking of a message transmission operation until the beginning of data injection into the channel. Function of the message size, $o(m)$, is the sum of software stack O_S and the protocol O_P. It is a step function with threshold H, so that $o(m) = O_S$ if $m < H$, $o(m) = O_S + O_P$ if $m \geqslant H$. The implementation of shared memory collectives in the studied libraries *MPICH* and *Open MPI* do not use protocol activities: so, $O_P = 0$ in this paper. As regards O_S, it is significant only on transmissions of very tiny messages in shared memory, and hence this parameter can be ignored in practice most of the times.

γ_{op}: Represented as $\gamma_{op}(m,\tau)$, it is the time invested in operating *two* vectors of size m by operation op. The number of elements in m bytes depends on the type of data. Parameter γ depends on the number of simultaneous operations in course (τ), because of the need to access the channel to obtain the operands.

P: Number of processes involved in the MPI operation.

N: Number of different *channels* in the system. This paper deals only with the shared memory channel.

3.2. Modeling message transmissions

This section illustrates the application of τ-*Lop* on modeling the cost of message transmission, denoted as $T(m)$ and always applied to a contiguous message.[2] Fig. 6 illustrates how the transmission flows in s stages, a figure that depends on n, the number of transfers each segment needs to achieve the receive buffer, as well as on k, the number of segments the message is split into, so that $s = k + n - 1$. n depends on the location of processes, as well as on the software mechanisms used, which can vary with the message size or with the mode of communication. $m = S \times k$, S being the constant size of the segment in the implementation. It is to be noted that k equals 1 in non-segmented transmissions. For instance, *MPICH-KNEM* sends a long message through in a single transfer ($n = 1$), and it sends a not so long message, by segmenting it, through a shared intermediate buffer ($n = 2$). The total number of transfers in a message transmission is $k \times n$. Left side of Fig. 6 shows a single transfer transmission, with $n = 1$ and $k = 1$. Right side shows transmission of the same message broken into $k = 3$ segments. As $n = 2$, there are a total of $k \times n = 6$ transfers and hence $s = 4$ stages. It is to be noted how first and last transfers proceed alone, while the other four proceed in concurrent pairs. Thus, τ-*Lop* defines $T(m)$ as a two-term sum, the overhead and the aggregated cost of the individual transfers:

$$T(m) = o(m) + \sum_{j=0}^{s-1} L_j(m/k, \tau_j) \tag{9}$$

For example, in *MPICH*, where $n = 2$, $s = k + n - 1 = k + 1$. From now on $o(m)$ will be considered negligible, and so it will not appear in further derivations, making (9) to become

$$T(m) = 2L(S,1) + (k-1)L(S,2) \tag{10}$$

A particular case of (10) happens when the message size is smaller than segment size ($m < S$). Here, $k = 1$, and so $s = 2$. Now (9) becomes

$$T(m) = 2L(m,1) \tag{11}$$

It is to be noted that (11) equals the cost given by $\log_n P$ in (5); so, $\log_2 P$ is a particular case of τ-*Lop* when $n = 2$ and $k = 1$, namely, when τ-*Lop* ignores the segmentation. Another example is *MPICH-KNEM*, where $n = 1$, and $k = 1$ owing to lack of segmentation (see left side of Fig. 6). It results in $s = 1$, and then (9) becomes $T(m) = L(m,1)$.

$\log_n P$ ignores the internal structure of the transmission and hence $T(m)$ becomes a black-box that has to be measured for every value of m. The only parameter in (5) is the overhead o, which is directly calculated as half of a *MPICH* ping, hence the estimation error of a point-to-point message in $\log_n P$ is simply zero. While $\log_n P$ measures $T(m)$, τ-*Lop* estimates it as the suitable composition of the $L(S,\tau)$ terms (see Fig. 6), which makes a difference. Fig. 7 compares models (10) and (11) with

[2] Under τ-*Lop* the cost of packing and unpacking a non-contiguous message is attributed to the operation, so that, for example, $\Theta_{p2p}(m) = 2p(m) + T(m)$.

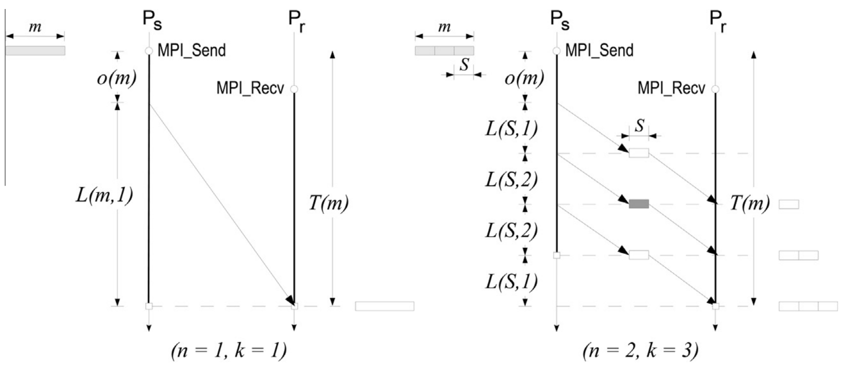


Fig. 6. Representation of the parameters of the model τ-*Lop* through the shared memory communication channel including overhead. On the left side the message is sent without segmentation, and on the right side the message is sent as three segments of size *S*. The transfers are done through an intermediate buffer of two components of the same size.

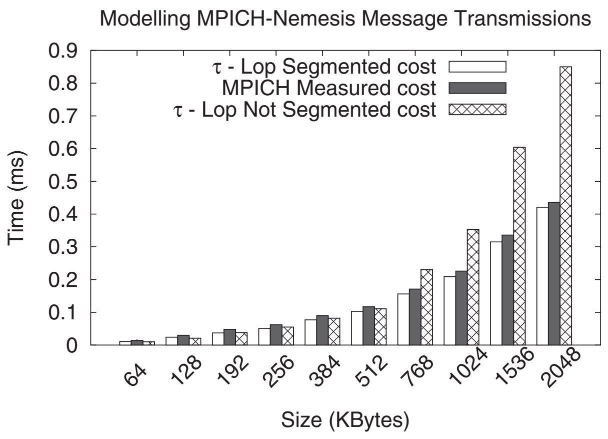


Fig. 7. *MPICH* empirically measured point-to-point latency compared to its segmented and non-segmented modeling with τ-*Lop*. The target machine is *Metropolis*, a two-socket Intel Xeon E5620 (Nehalem) (see Section 5).

respect to the real measured cost of *MPICH* in shared memory. The message size m ranges from 64 KB to 2 MB, producing an average relative error of 13.8% in (10), while achieving 30.2% using (11). The overlapped interchange of segments between sender and receiver reduces the transmission cost, an effect that τ-*Lop* captures with reasonable accuracy.

3.3. Concurrent transmissions

The concurrency of transfers has its impact on the cost of concurrent transmissions. If the cost of the message transmission between processes 0 and 8 in Fig. 1 is $T(m)$, the cost between processes 14 and 15 is higher because eight concurrent transmissions share the memory bandwidth. In contrast, the costs are the same with $\log_n P$ and *LogGP*, which is unrealistic. The true cost between processes 14 and 15 ranges between $T(m)$, for a perfectly parallel channel, and $8 \times T(m)$ for a serial channel. The $\|$, operator is introduced to represent this fact. The expression $A\|L(m,\tau)$ represents the cost of A concurrent sets of in turn τ concurrent transfers. It is defined as $A\|L(m,\tau) = L(m, A \times \tau)$. Fig. 8 illustrates the concept. It shows the pair of message transmissions ($A = 2$) taking place at the second stage of Fig. 1. It is defined that $A\|(L_1(m,\tau) + L_2(m,\tau)) = A\|L_1(m,\tau) + A\|L_2(m,\tau)$, so that concurrency operator $\|$ possesses the distributive property with respect to the addition (sequence) of the costs of serial transfers. As a result, $\|$ can be applied to describe the cost of A concurrent message transmissions as $A\|T(m)$, defined as follows:

$$A\|T(m) = A\|\sum_{j=0}^{s-1} L_j(m,\tau_j) = \sum_{j=0}^{s-1} A\|L_j(m,\tau_j) = \sum_{j=0}^{s-1} L_j(m, A \times \tau_j) \quad (12)$$

4. Modeling MPI collectives with τ-*Lop*

This section puts forward the τ-*Lop* modeling of some algorithms used to implement *MPI_Bcast* on shared memory in *MPICH* and *Open MPI*. Also, how they can be analytically compared is illustrated with an example.

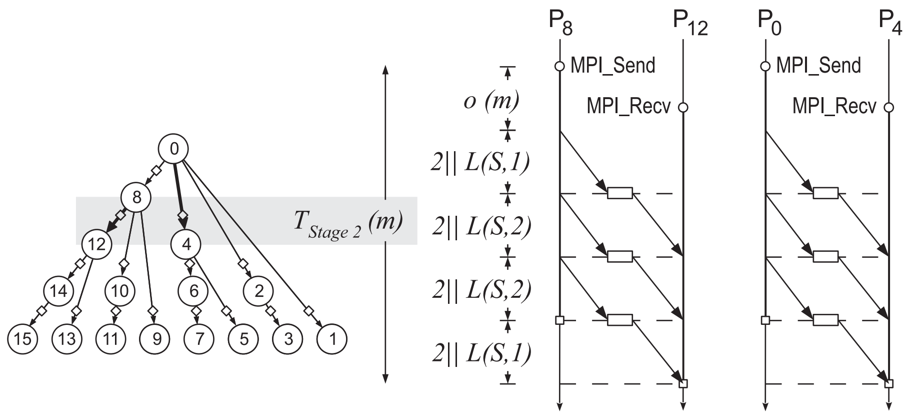


Fig. 8. τ-*Lop* cost of stage 2 of a binomial broadcast, expressed in terms of concurrent transfers.

4.1. The MPICH broadcast algorithms

The algorithms used in *MPICH* to implement the *MPI_Bcast* primitive are representative, because they are also used as building blocks by other MPI collectives, such as *MPI_Scatter*, *MPI_Allgather* and *MPI_Alltoall*. The *binomial tree* algorithm is used for instance for short messages and a small number of processes ($m < 12$ KB and $P < 8$). For regular messages ($12\ \text{KB} \leqslant m < 512\ \text{KB}$) and power-of-two P, however, *scatter*, followed by *recursive doubling allgather (RDA)* algorithm, are used, so that the *MPI_Bcast* cost is the sum of the two individual costs,[3] $\Theta_{Scatter}(m) + \Theta_{RDA}\left(\frac{m}{P}\right)$. For other message sizes, the *ring* algorithm replaces *RDA*.

These four algorithms are of further interest because they have contrasting features. In the *binomial tree* broadcast algorithm, the number of involved processes grows with successive stages, while the message size remains constant. In the *scatter* algorithm, the number of involved processes grows, while the message size halves. In the *RDA* algorithm, the number of involved processes remain constant while the message size doubles. Finally, in the *ring* algorithm both the number of communicating processes and the size of the interchanged messages remains constant.

The binomial tree algorithm: The height of a binomial tree of P processes is $h(P) = \lceil \log_2(P) \rceil$, and hence it requires $\lceil \log_2(P) \rceil$ stages, as shown in Fig. 1. The cost of the first stage equals the cost of a message transmission $T(m)$. The cost of subsequent stages is expressed by (12) as $2\|T(m),\ 4\|T(m),\ 8\|T(m)$ and so on. Hence, the cost of the whole broadcast on a full tree would be as follows:

$$\Theta_{Bin}(m) = \sum_{i=0}^{\lceil \log_2(P) \rceil - 1} \left(2^i \| T(m)\right) \tag{13}$$

If $T(m)$ is given by (10), then (13) becomes as follows:

$$\Theta_{Bin}(m) = \sum_{i=0}^{\lceil \log_2(P) \rceil - 1} \left(2^i \| (2L(S,1) + (k-1)L(S,2))\right) = \sum_{i=0}^{\lceil \log_2(P) \rceil - 1} \left(2L(S,2^i) + (k-1)L(S,2^{i+1})\right) \tag{14}$$

On the contrary, if $T(m)$ is given by (11), the broadcast cost becomes thus:

$$\Theta_{Bin}(m) = \sum_{i=0}^{\lceil \log_2(P) \rceil - 1} \left(2^i \| (2L(m,1))\right) = 2 \sum_{i=0}^{\lceil \log_2(P) \rceil - 1} L(m,2^i) \tag{15}$$

Fig. 9 plots the relative error made by τ-*Lop* and $\log_n P$ with respect to the measured cost of the *binomial tree* broadcast algorithm (13), in *Metropolis*. The machine had an L2 cache of 256 KB and an L3 cache of 12 MB, figures that leave their footprint in the *MPI_Bcast* bandwidth despite the intent of the *-off_cache* IMB option to avoid the cache effect, which substantially increases the τ-*Lop* error in the range of messages which fits in the cache hierarchy. *MPICH* was modified to use the *binomial tree* algorithm for all message sizes. In addition, the default segment size of 64 KB was change to 8 KB. The library had to be configured with the *-disable-smpcoll* option and rebuilt. Version 3.2.3 of *IMB* deploys the broadcast scenery, setting the root to process number 0. Fig. 10 reproduces the former test for increasing P in *Lusitania*, which shows that the τ-*Lop* mean error decreases asymptotically with P. The $\log_n P$ error, however, ends up growing with P, because of ignoring the effect of concurrent transfers.

[3] While T is commonly used to represent the cost of a single point-to-point transmission involving two processes, Θ is used to represent the cost of a whole collective algorithm.

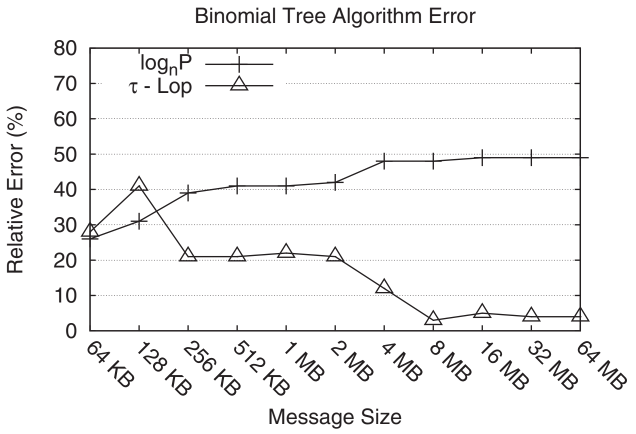


Fig. 9. *Binomial Tree* algorithm relative error made by *τ-Lop* and $\log_n P$ cost models with respect to the real cost of the *MPICH* implementation for eight processes and growing message size in *Metropolis*.

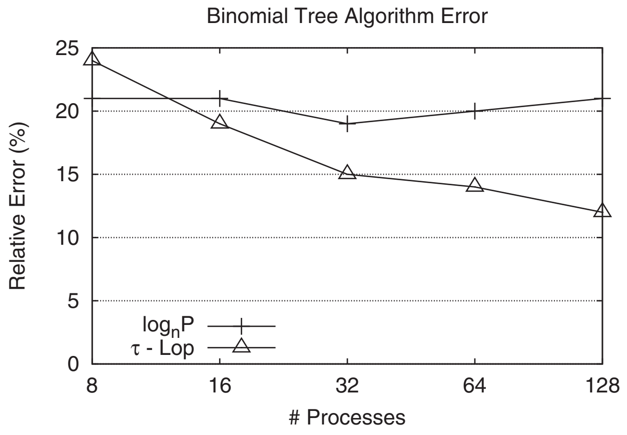


Fig. 10. *Binomial Tree* algorithm relative error compared to *MPICH* implementation for increasing the number of processes *P*, in *Lusitania* machine. The message size used in each *P* ranges between 8 KB and 32 MB.

Regarding *LogGP*, measuring its parameter in shared memory is difficult. A minimal variation, mainly in *L*, leads to a great prediction error. The tool used for the measurement [24] leads to more than 75% relative error for any *P*.

The Scatter and Recursive Doubling Allgather algorithms: The *Scatter* algorithm scatters a message of size *m*, across *P* processes, using a binomial tree in such a way that the size of the message halves at each stage till it achieves a value of m/P, the size finally received by every process. The operation involves the following cost:

$$\Theta_{Scatter}(m) = \sum_{i=0}^{\lceil \log(P) \rceil - 1} \left(2^i \| T\left(\frac{m}{2^{i+1}}\right) \right) \tag{16}$$

It is to be noted how the message size halves with each stage. The *Recursive Doubling Allgather* algorithm (*RDA*) is based on a *send-receive* operation (defined as *MPI_Sendrecv* in the MPI Standard). On it, a process p_i sends a message of size *m* to a process p_j and next receives another message of size *m* from process p_k. For *P* participants, the cost of the operation under *τ-Lop* will be $T_{Sendrecv}(m) = P\|2L(m,1) = 2L(m,P)$. For $m > S$, the segmented formulation will be applied with cost:

$$T_{Sendrecv}(m) = P\|2kL(S,1) = 2kL(S,P) \tag{17}$$

The cost of the *RDA* operation is:

$$\Theta_{RDA}(m) = \sum_{i=0}^{\log(P)-1} \left(P\|T_{Sendrecv}\left(2^i m\right) \right) \tag{18}$$

It is to be noted how the message size doubles with each stage. Estimations for *LogGP* and $\log_n P$ could be worked out from [7,19] respectively thus:

$$T_{RDA}^{LogGP} = (P-1)k \times (o + (S-1)G + max\{L+o, g\}) \tag{19}$$

$$T_{RDA}^{\log_n P} = \sum_{i=0}^{\log(P)-1} \left(2^{i+1} o\right) \tag{20}$$

Fig. 11 shows the relative error made by τ-*Lop* (18), *LogGP* (19) and $\log_n P$(20) cost estimations of the *RDA* algorithm, for increasing P, with respect to the real cost measured in *MPICH*. *LogGP* has been considered incapable of predicting shared memory performance. Its error, however, is not as high as expected, although higher than that of τ-*Lop*, again with figures that remain constant when P increases.

The Ring Allgather algorithm: For the rest of message sizes and non-power-of-two P, the broadcast operation of *MPICH* uses the same *Scatter* algorithm modeled by (16), followed by an *allgather* stage, this time implemented with a ring algorithm. It includes $P-1$ stages, each stage with P simultaneous *send–receive* operations (17) in a ring, moving a message of constant size:

$$\Theta_{RingAllgth}(m) = (P-1) \times (P \| T_{sendrecv}(m)) = 2k(P-1)L(S,P) \tag{21}$$

Although accuracy is important in a cost prediction model, its expressive power is even more important. It is to be noted how a complex algorithm, such as *Ring Allgather*, is modeled by τ-*Lop* in (21) in a compact and expressive way. *LogGP* and $\log_n P$, on the contrary, produce no meaningful formulations, because they simply multiply definitions (2) and (5) by $(P-1)$ stages. The application of τ-*Lop* shows a smaller mean relative error than that shown by $\log_n P$or *LogGP*, as can be seen in Fig. 12.

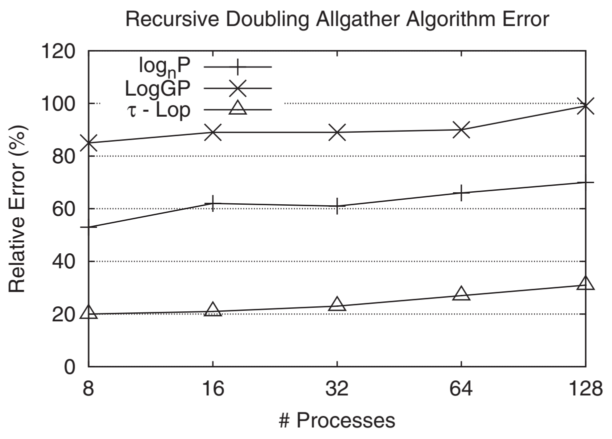


Fig. 11. Relative error of three performance models of *Recursive Doubling Allgather* algorithm, with respect to the real cost of its *MPICH* implementation in Lusitania machine. The message size m ranges between 8 KB and 32 MB on an increasing number of processes P involved in the operation.

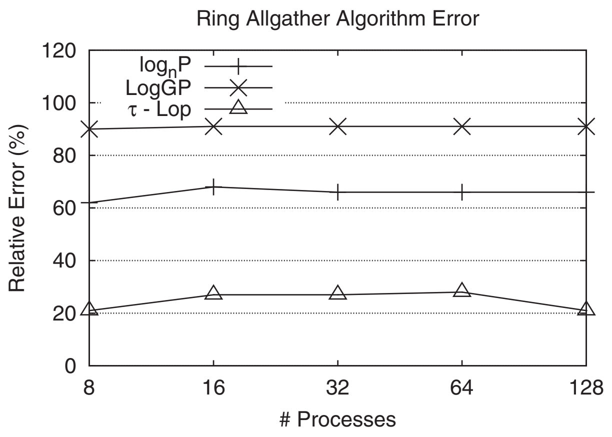


Fig. 12. Relative error of *Ring Allgather* algorithm under the same conditions as those of Fig. 11.

4.2. The Open MPI broadcast algorithm

The *Open MPI* algorithm is different, because it is not built based on message transmissions. Another basic construct, the collective transfer, is used instead. Unlike *MPICH*, it is applied to all the message sizes. It operates as a tree of degree g (known as *binary* tree for $g = 2$), instead of a *binomial* tree. The height of a tree of P processes is $h(P) = \lceil \log_g((g-1)P+1) \rceil$. Level i has g^i processes. Data communication is not based on message transmissions between pairs of processes, but on using a memory zone which is shared by all the processes involved in the operation. Specifically, a parent process broadcasts data to its g children through an intermediate buffer, composed by segments of $S = 8$ KB. Look at Fig. 13, a non-full tree of degree $g = 2$, with $P = 8$ processes, user buffers of $k = 4$ segments, and intermediate two-segment buffers. Given a parent process P_p and a child P_c, child P_c copies every segment from his father's intermediate buffer b_p, first to its own intermediate buffer b_c, and then to its own user buffer u_c. The idea of this approach is that parent and child transfers progress in parallel in pipeline. Thus, child transfers from P_1 to P_3 and from P_1 to P_4 progress, with some delay, in parallel with the parent transfers from P_0 to P_1.

As the concept of transmission disappear from the algorithm, cost terms T vanish from its τ-*Lop* model, and this has to be formulated directly in terms of the lower level transfer costs L, a change that does not contribute to its expressiveness. Nevertheless, the algorithm has a pipeline behavior, fully detailed in Fig. 14. Three sections can be identified in the figure, the head, the body and the tail of the pipeline, with respective costs $\Theta_H,\ \Theta_B$ and Θ_T, so that $\Theta_{Ompi}(m) = \Theta_H + \Theta_B + \Theta_T$.

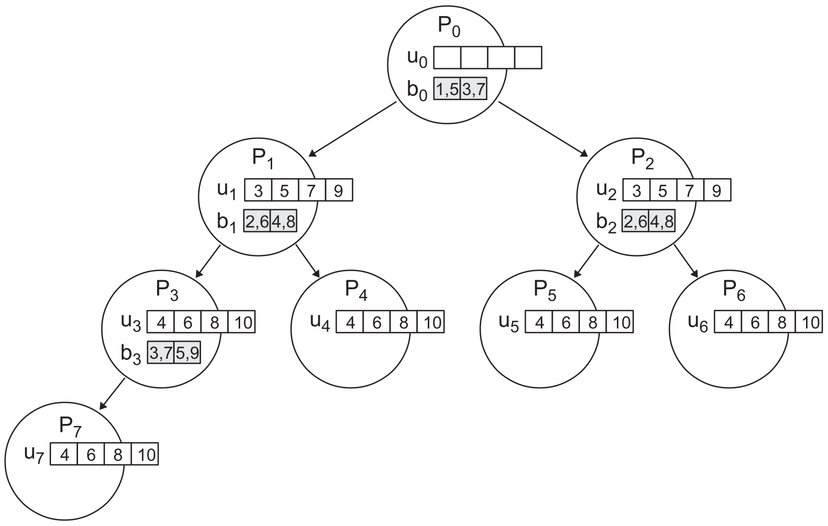
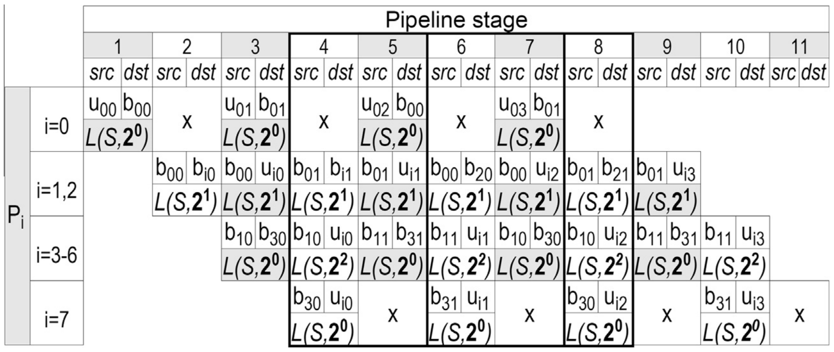


Fig. 13. Buffering layout of the *Open MPI COLL SM* Broadcast algorithm in a non-full binary tree of $P = 8$. Each involved process P_i has its user buffer u_i divided in $k = 4$ segments $u_{ij},\ 0 \leqslant j < k$. In addition, it owns an intermediate buffer b_i (in gray) of two segments $b_{ij},\ 0 \leqslant j < 2$, in shared memory, used to communicate to his children. Numbers inside the segments are temporal labels. Label n denotes that the segment is written in the pipeline step n. Segment b_{00}, for instance, is written in stage 1 and again in stage 5. Segments $b_{01},\ u_{10},\ u_{20}$ and b_{30} are written concurrently at stage 3.



		Pipeline stage																					
		1		2		3		4		5		6		7		8		9		10		11	
		src	dst	src	dst	src	dst	src	dst	src	dst	src	dst	src	dst	src	dst	src	dst	src	dst	src	dst
P_i	i=0	u_{00}	b_{00}	X		u_{01}	b_{01}	X		u_{02}	b_{00}	X		u_{03}	b_{01}	X							
		$L(S,2^0)$				$L(S,2^0)$				$L(S,2^0)$				$L(S,2^0)$									
	i=1,2			b_{00}	b_{i0}	b_{00}	u_{i0}	b_{01}	b_{i1}	b_{01}	u_{i1}	b_{00}	b_{20}	b_{00}	u_{i2}	b_{01}	b_{21}	b_{01}	u_{i3}				
				$L(S,2^1)$		$L(S,2^1)$		$L(S,2^1)$		$L(S,2^1)$		$L(S,2^1)$		$L(S,2^1)$		$L(S,2^1)$		$L(S,2^1)$					
	i=3-6					b_{10}	b_{30}	b_{10}	u_{i0}	b_{11}	b_{31}	b_{11}	u_{i1}	b_{10}	b_{30}	b_{10}	u_{i2}	b_{11}	b_{31}	b_{11}	u_{i3}		
						$L(S,2^0)$		$L(S,2^2)$		$L(S,2^0)$		$L(S,2^2)$		$L(S,2^0)$		$L(S,2^2)$		$L(S,2^0)$		$L(S,2^2)$			
	i=7							b_{30}	u_{i0}	X		b_{31}	u_{i1}	X		b_{30}	u_{i2}	X		b_{31}	u_{i3}	X	
								$L(S,2^0)$				$L(S,2^0)$				$L(S,2^0)$				$L(S,2^0)$			

Fig. 14. Evolution of the pipeline produced by Fig. 13. Time runs to the right. Each cell shows the source and destination buffers of the specific transfer in the top half, and its cost in the bottom half. The cost of stage 4, for instance, is $L(S,2^1)\|L(S,2^2)\|L(S,2^0) = L(S,2^1+2^2+2^0) = L(S,7)$ (note that the stage label 4 appears seven times in Fig. 13). The cost of the operation is the sum of the costs of all the stages.

An analytical transfer-based expression of the cost can be obtained from this pattern with τ-*Lop*, though not necessarily as directly and elegantly as from the former transmission-based expressions. It can be shown, for instance, that if $g = 2$, the body is a repetition of $k - 2$ two-stage equal blocks when $log_2 P$ is even. Hence $\Theta_B = (k - 2)(\Theta_{B_0} + \Theta_{B_1})$, where Θ_{B_0} and Θ_{B_1} are the costs of the first and second stages of the block. A stage Θ_{B_0} has to be added when $log_2 P$ is odd (as is the case of Fig. 14); so, $\Theta_B = (k - 2)(\Theta_{B_0} + \Theta_{B_1}) + (log_2 P \bmod 2)\Theta_{B_0}$. Fig. 14 illustrates that, on a full tree, $\Theta_{B_0}(m) = \|_{i=0}^{h(P)-1} L(S, g^i)$ and $\Theta_{B_1}(m) = \|_{i=1}^{h(P)} L(S, g^i)$.

As regards $\log_n P$ and *LogGP*, transmission-based models, they just cannot represent the cost of this algorithm. As an approximation, $\log_n P$ could model it as $T_{SM}^{\log_n P} = h(P) \times (2o)$, simply by multiplying the height of the binary tree with the cost of a point-to-point message. The *LogGP* cost would be $T_{SM}^{LogGP} = h(P) \times (2o + L + (S - 1)G + (k - 1)(g + (S - 1)G))$. Of course, neither of the two formulas can draw a picture of the concurrent pipeline behavior of the algorithm. Fig. 15 shows, once again, the relative error of the models. For some reason, the performance of the *Open MPI* implementation of the algorithm degrades for higher number of processes ($P > 64$), which leads to a noticeable error on τ-*Lop*. It is also to be noted that with $\log_n P$, the error varies erratically with P, while τ-*Lop* remains quite stable. The $\log_n P$ results are, therefore, considered unreliable.

4.3. Reasoning with τ-Lop

Fig. 16 plots the cost of a message transmission in a repetitive sequence. In this iterative scenery, the cost of message transmission (10), rather than *T(k) = 2L(S,1) + (k-1) L(S,2)*, must be modeled as *T(k) = k L(S,2)*, because the first transfer of the iteration *t* gets overlapped by the last transfer of the iteration *t-1* and, similarly, the last transfer of *t* gets overlapped by the first transfer of *t* + 1. Hence, one can adopt the following as the cost of a message transmission:

$$T(m) = kL(S, 2) \tag{22}$$

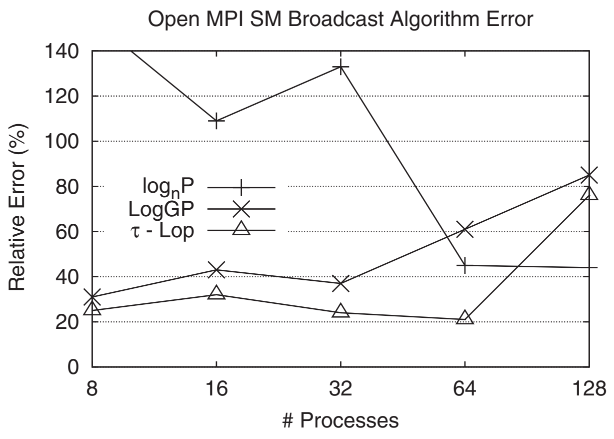


Fig. 15. Relative error of the *Open MPI COLL SM Broadcast* algorithm under the same conditions as those of Fig. 11.

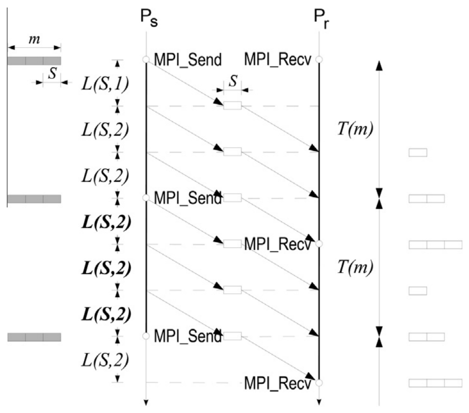


Fig. 16. τ-*Lop* cost modeling of sequential message transmissions.

As (10) nearly equals (22) for great k, using (22) is a reasonable approximation. By virtue of being a simpler construct, it produces simpler cost expressions of collective algorithms. The former algorithms proceed in a sequence of stages that fit the conditions of application of (22), and so their simplified new formulations should not lead τ-*Lop* to any loss of accuracy, but rather to the opposite. Based on these premises, the τ-*Lop* predictions of *Scatter* (16) and *RDA* (18), can be given thus for the case of segmentation:

$$\Theta_{scatter}(m) = \sum_{i=0}^{\lceil\log(P)\rceil-1}\left(2^i\|\left(\frac{P}{2^{i+1}}\right)kL(S,2)\right) = \sum_{i=0}^{\lceil\log(P)\rceil-1}\left(\frac{P}{2^{i+1}}kL(S,2^{i+1})\right)$$

$$\Theta_{RDA}(m) = \sum_{i=0}^{\log(P)-1}\left(P\|\left(2^{i+1}kL(S,1)\right)\right) = \sum_{i=0}^{\log(P)-1}\left(2^{i+1}kL(S,P)\right)$$

Deploying the summatory functions gives the following:

$$\Theta_{scatter}(m) = k\left(\frac{P}{2}L(S,2) + \frac{P}{4}L(S,4) + \frac{P}{8}L(S,8) + \ldots + L(S,P)\right) \tag{23}$$

$$\begin{aligned}\Theta_{RDA}(m) &= k(2L(S,P) + 4L(S,P) + 8L(S,P) + \ldots + PL(S,P)) = 2k\left(L(S,P) + 2L(S,P) + 4L(S,P) + \ldots + \frac{P}{2}L(S,P)\right)\\ &= 2k\left(\frac{P}{2}L(S,P) + \frac{P}{4}L(S,P) + \frac{P}{8}L(S,P) + \ldots + L(S,P)\right)\end{aligned} \tag{24}$$

Section 2 discusses why *LogGP* and $\log_n P$ equal the cost of *RDA* and *scatter*. On the contrary, τ-*Lop* derivations (23) and (24) reveal a noticeable formal difference, which helps to explain why *RDA* is between 2.5 and 3.5 times more expensive than *scatter* for any m and P, as Fig. 17 shows. Indeed, in each stage, an *RDA* transfer is performed by all the P processes concurrently, probably exhausting the memory bandwidth, while in *scatter*, the number of processes accessing memory increases with each stage. The capability of τ-*Lop* for analytical inference is another valuable accomplishment of the model. For instance, in the light of (23) and (24), it can be reasoned as follows. To begin with, $\Theta_{scatter}(m)$ has a first factor of value k, which doubles in $\Theta_{RDA}(m)$. The second factor is also greater on $\Theta_{RDA}(m)$, because the coefficients of the L terms are identical in both expressions, but the L terms themselves are all $L(S,P)$ in $\Theta_{RDA}(m)$ and they only increase up to $L(S,P)$ in $\Theta_{scatter}(m)$. As a result, τ-*Lop* produces a comparative cost expression that agrees with the experimental results of Fig. 17:

$$\begin{aligned}\Theta_{RDA}(m) &= 2k\left(\frac{P}{2}L(S,P) + \frac{P}{4}L(S,P) + \frac{P}{8}L(S,P) + \ldots + L(S,P)\right) > 2k\left(\frac{P}{2}L(S,2) + \frac{P}{4}L(S,4) + \frac{P}{8}L(S,8) + \ldots + L(S,P)\right)\\ &> 2\Theta_{scatter}(m)\end{aligned}$$

5. The estimation of $L(S,\tau)$

The effective application of τ-*Lop* poses the problem of estimating $L(m,\tau)$ for every τ and m on the target machine, a problem that this section addresses. For this work, two ccNUMA platforms of contrasting features are used: *Lusitania* and

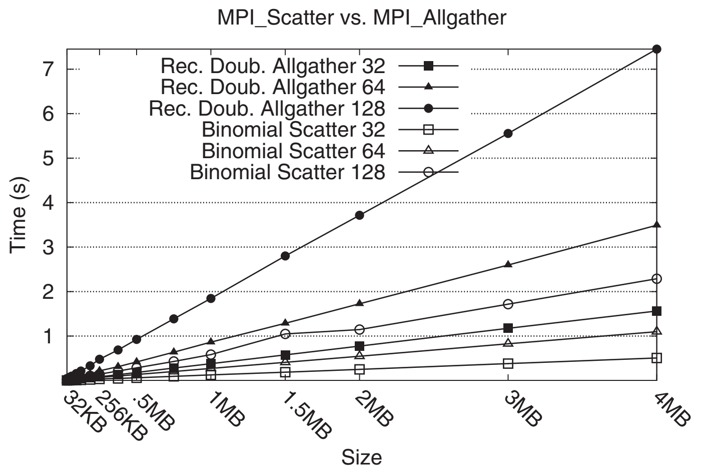


Fig. 17. Comparison of measured costs of *binomial scatter* and *RDA* algorithms with different P values (32, 64 and 128) and increasing m message size. The experiment was conducted on the shared memory computer *Lusitania*, an HP Integrity Superdome SX2000, composed of 128 cores (see Section 5).

Metropolis. *Lusitania* is a system with two HP Integrity Superdome SX2000 cabinets installed at Extremadura Supercomputing Center (CénitS) in Cáceres, Spain. The two cabinets are connected by 10 GB Ethernet. Each individual cabinet served as a target shared-memory machine, equipped with 64 Dual-Core Intel Itanium2 Montvale 9140 N processors (running at 1.4 GHz). That made up a total of 128 cores, split into 16 blade cells (NUMA nodes). Each cell has 8 cores with private L3 cache of 9 MB (unlike *Metropolis*, there is no shared cache in *Lusitania*). The operating system is GNU/Linux 2.6.16 and the compiler used is *GCC* version 4.1.2. With 1 Terabyte of main memory shared by 128 cores, Lusitania is one of the biggest shared-memory computers of Europe. An outstanding feature of Lusitania is its uniform performance, i.e., intra- and inter-cell latency are virtually the same. This parity in latency is manifest in the fact that the costs of intra- and inter-NUMA message transmissions being equal. The present results show, for instance, that a binomial broadcast between eight processes bound to the eight cores of NUMA 0 achieves the same performance when bound to the core 0 of eight different NUMAs of the machine. *Metropolis*, on the other hand, is a conventional commodity two-socket Intel Xeon CPU E5620 (Nehalem) running at 2.4 GHz. Each socket (NUMA node) has four cores, each with an L2 cache of 256 KB, that share an L3 cache of 12 MB, thus providing a machine with a total of 8 cores. The operating system was GNU/Linux 2.6.32, and the compiler used *GCC* version 4.4.6. Inter-NUMA latency was greater than the intra-NUMA counterpart, as is always the case with this type of platforms.

Current memory systems are faster for reading than for writing. This difference is surprising, because from the perspective of the physical DRAM memory module, load and store operations take approximately the same amount of time. Writes take about twice as long as reads, because of prefetching and the way the cache is integrated into the memory subsystem. Benchmarks show that the read bandwidth overcomes the write bandwidth by a factor of 1.5 in the HP Integrity Superdome. Similarly, according to [25], the read bandwidth doubles the write bandwidth in the Nehalem architecture, a figure confirmed by our tests. The authors' tests show that the read bandwidth overcomes the write bandwidth by a factor of 1.5 in the HP Integrity Superdome. Similarly, according to [25], the read bandwidth doubles the write bandwidth in the Nehalem architecture, a figure confirmed by the authors tests. This means that the cost $L(S,\tau)$ of transferring a segment from its user send buffer to the intermediate buffer (read) would be half the cost from the intermediate buffer to the user receive buffer (write). Introducing these hardware dependent issues in *τ-Lop* would make the model over-detailed, and hence difficult to understand and use. However, keeping *τ-Lop* simple entails a degree of indetermination in the characterization of $L(S,\tau)$.

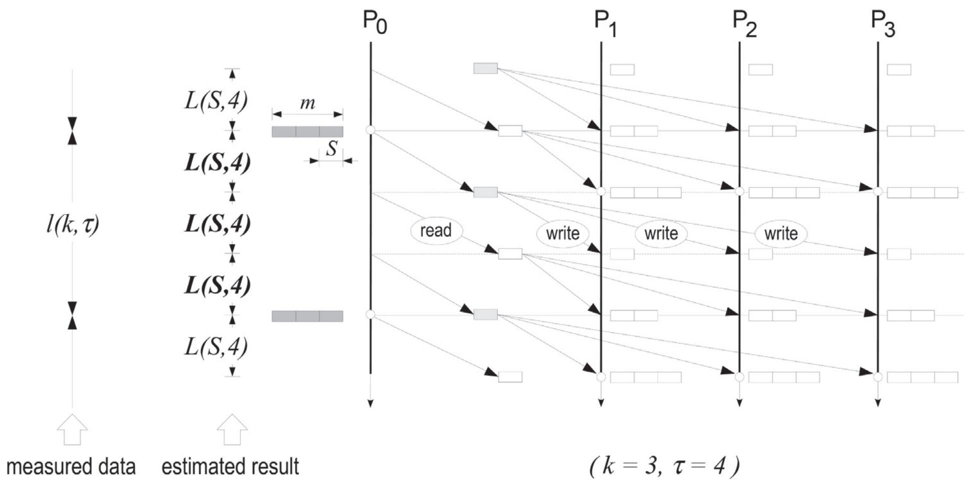
A suitable experimental design is essential to perform any parameter estimation, and to this end a software tool, namely *tauLop* [26], was developed. It produces a variety of concurrent transfer setups, measures their performance and calculates $L(m,\tau)$ from the obtained data. Each adopted setup, however, leads to different estimations, and so three different approaches are discussed next. To guide the exposition, the goal was supposed to be the determination of $L(8\ \text{KB},4)$, using segments of size $S = 8$ KB.

The first setup was based on Fig. 18a, a broadcasting between τ processes ($\tau = 4$), with two intermediate segments. *tauLop* creates this specific scenery (see Fig. 18b) by invoking the command "*$./tauLop -bcast -off_cache -local 2 8 4*". One needs to note the *-bcast* parameter. Parameter *-local* binds processes to cores, so that P_0, acting as root, is bound to core 0 of NUMA 0. Non-root processes first fill the rest of the cores of NUMA 0, and then the NUMAs 1, 2, 3, etc. The pair of parameters [2,8] after *-local* means that the transfers are performed through an intermediate buffer of two segments, each of size $S = 8$ KB. The last parameter 4 starts four threads. Finally, the parameter *-off_cache* keeps the send and receive user buffers out of cache, with the goal of determining L as the cost of a transfer between main memory (user buffer) and L3 cache (intermediate segments). Both send and intermediate buffers are allocated in the NUMA 0. Each receiving buffer was allocated to the NUMA where its corresponding process runs. The key issue of this setup is that Fig. 18a states that $l(k,\tau) = kL(S,\tau)$, so that $L(S,\tau) = l(k,\tau)/k$. This broadcast scenery should perform as an upper bound on the estimation of $L(S,\tau)$. It is to be noted that it consists of τ processes engaged on a collective transfer across NUMAs. The $\tau - 1$ simultaneous copies from the shared intermediate segments in NUMA 0 to the $\tau - 1$ receiving user buffers in NUMA 0, 1, 2, etc, end up exhausting the bandwidth of NUMA 0. Effectively, $L(S,\tau)$ grows linearly with τ having a steep slope. It is to be noted that this broadcast technique is inefficient and hence not adopted for collective operations in current MPI implementations, that opt for hierarchical arrangements.[4] In other words, it characterizes the behavior of *τ-Lop* under these rather extreme conditions.

The second approach to estimate $L(S,\tau)$ was based on a pair of concurrent message transmissions ($\tau = 4$) (see Fig. 19), started by the command "*$./tauLop -ping -off_cache -local 2 8 4*". This time, the figure states that $l(k,\tau) = kL(S,\tau)$ for every k, so that $L(S,\tau) = l(k,\tau)/k$. The couples of processes first fill the NUMA 0 and then populate successive NUMA nodes. All buffers were allocated to the NUMA where their corresponding couple runs. On this occasion, no inter-NUMA communication takes place, again an unrealistic situation, this time at the opposite extreme to *-bcast*. Another problem with *-ping* arrangement is that it lacks synchronization between couples. As a result, it becomes impossible to assure that couples run synchronized along the time frame of Fig. 19, and thus the measurements are invalidated.

The third configuration was based on a ring of four processes ($\tau = 4$) created by the command "*$./tauLop -ring -off_cache -local 2 8 4*" (Fig. 20). One needs to note the parameter *-ring*. The figure states that $l(k,\tau) = 2kL(S,\tau)$ for every k, so that $L(S,\tau) = l(k,\tau)/2k$. Synchronization on the access to main memory is ensured by each pulsation of the ring: a transfer from

[4] The broadcast of Fig. 1 is an example. Let it be supposed that processes 0–7 run in the NUMA 0 and processes 8–15 in NUMA 1. Once the inter-NUMA message transmission of stage 1 is done, both NUMAs work in parallel with just internal communication.



(a) Broadcast setup

```
       k              m          l(k,   4)     L( 8K,   4)
  (Segments)       (Bytes)        (ns)          (ns)
  -----------  ------------  ------------  -----------

           1           8192          3793          3793
           2          16384          7666          3833
         ...            ...           ...           ...
        8192       67108864      31205259          3809
       12288      100663296      47056028          3829
  -----------------------------------------------------
                                        L(8KB, 4)  = 3794
```

(b) Textual output of the *taulop* command

Fig. 18. The *broadcast* approach to the estimation of $L(S,\tau)$. The figure illustrates the case $\tau = 4$. Messages of three segments were used ($k = 3$). One needs to note that $l(k,4)$ was measured in successive iterations (the average was taken). $L(S,4)$ was obtained from the hypothesis of $l(k,4) = kL(S,4)$, so that $L(S,4) = l(k,4)/k$. This setup was created by the command "./taulop -bcast -off_cache -local 2 8 4", whose output is shown. For better precision, $L(S,\tau)$ was estimated for growing k (the figure illustrates the case $k = 3$), and so the final $L(S,\tau)$ figure is the average of all them. (b) shows the output of the command, with a row per each value of k. Column l in the first entry, for instance, shows the cost of broadcasting a buffer of size m = 8 KB (or k = 1). It is really the average of a loop of 262,144 broadcasts, a number great enough to minimize the measurement error.

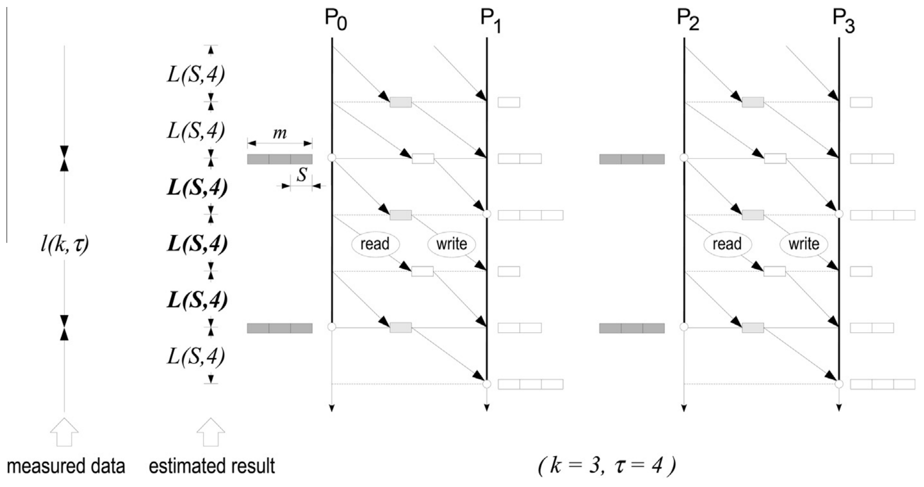


Fig. 19. The *ping* approach to the estimation of $L(S,4)$.

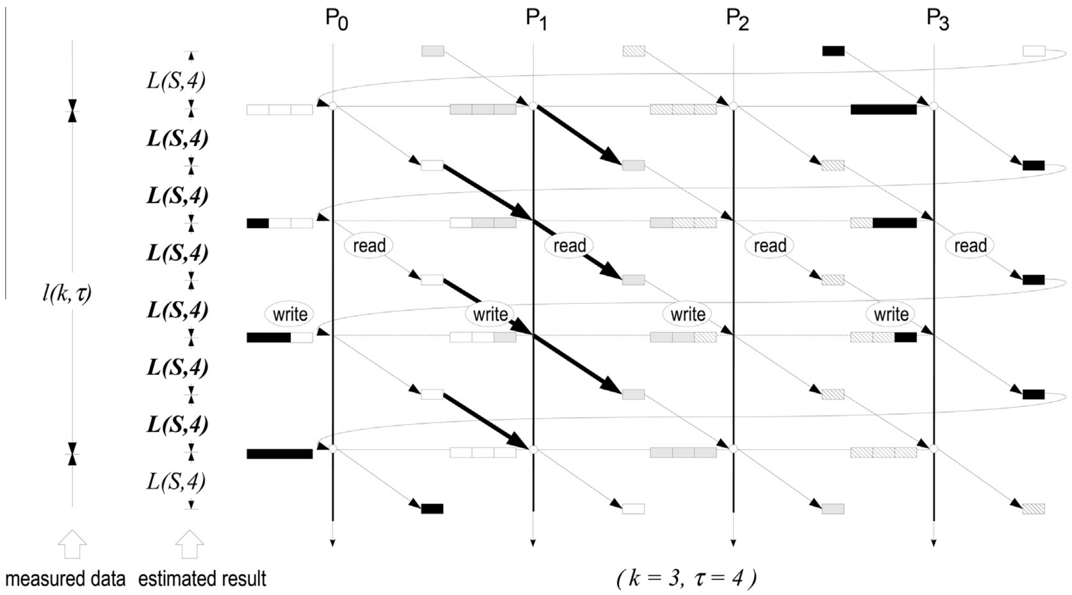
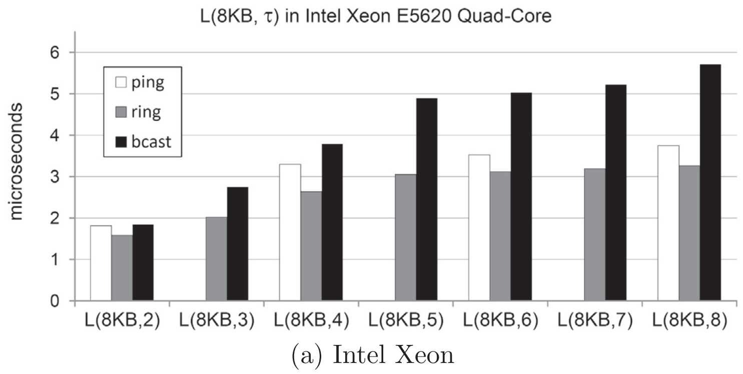


Fig. 20. The *ring* approach to the estimation of $L(S, 4)$.



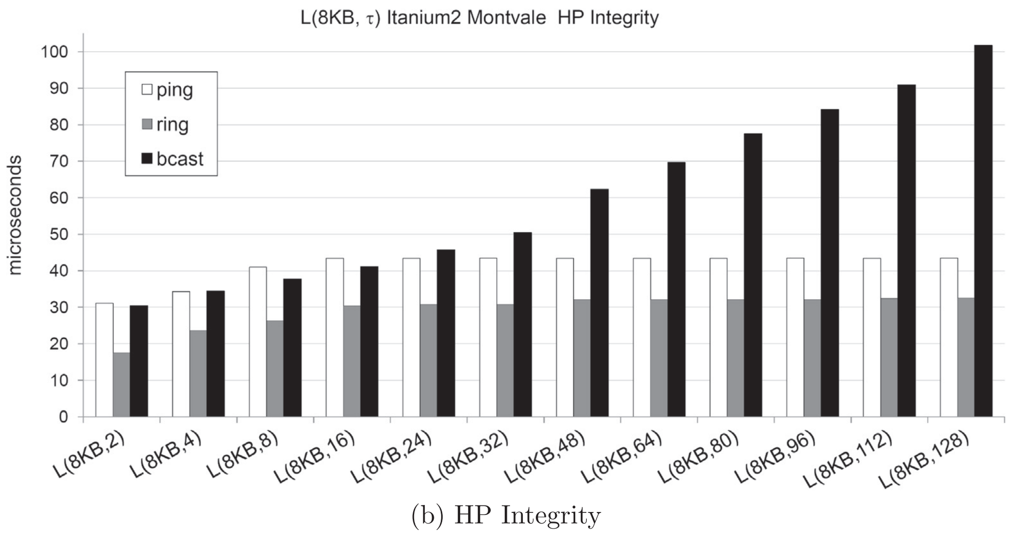


Fig. 21. Estimations of $L(S, \tau)$ in *Metropolis* and *Lusitania* machines under three considered approaches.

the user buffer to the intermediate segment shared with the right neighbor, followed by a second transfer from the intermediate segment shared with the left neighbor to the user segment just copied out.

The foregoing discussion on the application of the *tauLop* tool, for instance, has generated three different versions of $L(S,\tau)$, and many more are possible; so, which of them is the best of all? Fig. 21 shows the measurements of $L(8\,\mathrm{KB},\tau)$ obtained in the target platforms. Regardless of the obtained results, it seems evident that each choice favors the accuracy of some algorithms to the detriment of others. Indeed, our experimental work on the issue allows to draw some conclusions. For instance, it was detected here that the *-bcast* option achieves better accuracy on modeling the cost of *Open MPI SM* broadcast, because this algorithm is built upon the same type of collective transfers used in the *tauLop* tool (Fig. 18). As Fig. 21 shows, the *-bcast* estimations are too biased by the cited memory bandwidth exhausting problem, leading to a relative error that does not scale with *P* in most of collectives. Regarding *-ping*, its behavior is similar to that of *-ring*, and it leads to similar relative errors in collective costs, but *-ping* requires an even number of processes (see the missing bars in Fig. 21a), so it is less generic and hence discarded. The *-ring* approach offers a balanced mix of intra- and inter-NUMA communication and delivers estimations that have operated satisfactorily in our tests. Hence *-ring* is the method chosen to obtain $L(S,\tau)$.

6. Conclusions

This paper proposes τ-*Lop*, a formal model that predicts the cost of collective operations in shared memory, with focus on the operations defined in the MPI standard. The limitations of the precedent models in the multi-core arena were identified and the way by which they can be overcome with τ-*Lop* was demonstrated. The main contribution of τ-*Lop*, in the authors view, is its ability to model concurrent transfers in channels that are shared by all the processes implied in the communication. This concept makes possible the representation of sophisticated communication techniques used by current implementations of MPI in shared memory, such as collective transfers or message segmentation. As a result, one can obtain a more holistic picture of the cost expressions of collective operations, and hence a more precise model. The accuracy of τ-*Lop* was evaluated by dealing with a diversity of *MPICH* and *Open MPI* algorithms. The chosen algorithms cover a wide spectrum of the known collective communication techniques in shared memory. τ-*Lop* allows formal comparison of algorithms, because it is mathematically tractable. As an example, it has been formally demonstrated that the *binomial scatter* algorithm of *MPICH* is at least twice cheaper than *Recursive Doubling Allgather*, while previous models assign the same cost to them.

Extending τ-*Lop* to cover communication channels other than shared-memory, including networks of multi-cores, is the immediate challenge, and the experiments and modeling required in this regard are already under way. Besides, further research is needed on the parameter estimation of the model. The *tauLop* tool is just a seminal work on this issue, which needs further contributions, application in other platforms and, most important, new insights. Finally, the analyses of this paper were applied to messages that are large enough to ignore synchronization costs, and the efforts are now oriented to explore the capability of τ-*Lop* for modeling operations involving tiny messages, as also those of *MPI_Barrier*, where synchronization time represents the only cost.

Acknowledgments

The authors thank the Extremadura Supercomputing Center (CenitS) and High Performance Computing Center in Stuttgart (HLRS) for their support and for providing them the computational resources required for this study.

References

[1] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, T. von Eicken, Logp: towards a realistic model of parallel computation, in: Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP'93, ACM, New York, NY, USA, 1993, pp. 1–12.

[2] D.E. Culler, R.M. Karp, D. Patterson, A. Sahay, E.E. Santos, K.E. Schauser, R. Subramonian, T. von Eicken, Logp: a practical model of parallel computation, Commun. ACM 39 (1996) 78–85.

[3] A. Alexandrov, M.F. Ionescu, K.E. Schauser, C. Scheiman, Loggp: incorporating long messages into the logp model – one step closer towards a realistic model for parallel computation, in: Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA'95, ACM, New York, NY, USA, 1995, pp. 95–105.

[4] F. Ino, N. Fujimoto, K. Hagihara, Loggps: a parallel computational model for synchronization analysis, SIGPLAN Not. 36 (2001) 133–142.

[5] L. Yuan, Y. Zhang, Y. Tang, L. Rao, X. Sun, Loggph: a parallel computational model with hierarchical communication awareness, in: 2010 IEEE 13th International Conference on Computational Science and Engineering (CSE), pp. 268–274.

[6] C.A. Moritz, M.I. Frank, Logpc: modeling network contention in message-passing programs, IEEE Trans. Parallel Distrib. Syst. 12 (2001) 404–415.

[7] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G.E. Fagg, E. Gabriel, J.J. Dongarra, Performance analysis of MPI collective operations, Cluster Comput. 10 (2007) 127–143.

[8] M.P.I. Forum, MPI: a message passing interface standard, March, 1994.

[9] G.D. Benson, C. wai Chu, Q. Huang, S.G. Caglar, A comparison of MPICH allgather algorithms on switched networks, in: Proceedings of the 10th EuroPVM/MPI 2003 Conference, Springer, 2003, pp. 335–343.

[10] T. Hoefler, T. Schneider, A. Lumsdaine, Loggp in theory and practice: an in-depth analysis of modern interconnection networks and benchmarking methods for collective operations, Simul. Modell. Practice Theory 17 (2009) 1511–1521 (Advances in System Performance Modelling, Analysis and Enhancement).

[11] T. Hoefler, W. Gropp, W. Kramer, M. Snir, Performance modeling for systematic performance tuning, in: State of the Practice Reports, SC'11, ACM, New York, NY, USA, 2011, pp. 6:1–6:12.

[12] T. Hoefler, T. Schneider, A. Lumsdaine, Accurately measuring overhead, communication time and progression of blocking and nonblocking collective operations at massive scale, Int. J. Parallel Emerg. Distrib. Syst. 25 (2010) 241–258.

[13] R. Rabenseifner, Automatic profiling of MPI applications with hardware performance counters, in: Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer-Verlag, London, UK, 1999, pp. 35–42.
[14] R.W. Hockney, The communication challenge for MPP: Intel paragon and meiko cs-2, Parallel Comput. 20 (1994) 389–398.
[15] Argonne National Laboratory, MPICH project, <http://www.mpich.org>, 2013.
[16] 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, 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, pp. 97–104.
[17] D. Buntinas, G. Mercier, W. Gropp, Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem, in: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid, CCGRID'06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 521–530.
[18] K.W. Cameron, X.-H. Sun, Quantifying locality effect in data access delay: memory logp, in: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, IPDPS'03, IEEE Computer Society, Washington, DC, USA, 2003, p. 48.2.
[19] K.W. Cameron, R. Ge, X.H. Sun, $\log_m p$ and $\log_3 p$: accurate analytical models of point-to-point communication in distributed systems, Trans. Comput. IEEE 56 (2007) 314–327.
[20] B. Tu, J. Fan, J. Zhan, X. Zhao, Performance analysis and optimization of MPI collective operations on multi-core clusters, J. Supercomput. 60 (2012) 141–162.
[21] H. Zhu, D. Goodell, W. Gropp, R. Thakur, Hierarchical collectives in MPICH2, in: Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 325–326.
[22] H. wook Jin, S. Sur, L. Chai, D.K. Panda, Limic: support for high-performance MPI intra-node communication on linux cluster, in: International Conference on Parallel Processing ICPP, pp. 184–191.
[23] B. Goglin, S. Moreaud, KNEM: a generic and scalable kernel-assisted intra-node MPI communication framework, J. Parallel Distrib. Comput. 73 (2013) 176–188.
[24] T. Kielmann, H.E. Bal, K. Verstoep, Fast measurement of logp parameters for message passing platforms, in: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing, IPDPS'00, Springer-Verlag, London, UK, 2000, pp. 1176–1183.
[25] D. Molka, D. Hackenberg, R. Schone, M.S. Muller, Memory performance and cache coherency effects on an intel nehalem multiprocessor system, in: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques, PACT'09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 261–270.
[26] Media Engineering Group, taulop tool, <http://gim.unex.es/taulop>, 2013.