

# A Hyperbolic Model for Communication in Layered Parallel Processing Environments

Ion STOICA,<sup>\*</sup> Florin SULTAN<sup>\*</sup>, and David KEYES<sup>†</sup>

## Abstract

We introduce a model for communication costs in parallel processing environments, called the hyperbolic model, which generalizes two-parameter dedicated-link models in an analytically simple way. The communication system is modeled as a directed communication graph in which terminal nodes represent the application processes and internal nodes, called communication blocks (*CBs*), reflect the layered structure of the underlying communication architecture. A *CB* is characterized by a two-parameter hyperbolic function of the message size that represents the service time needed for processing the message. Rules are given for reducing a communication graph consisting of many *CBs* to an equivalent two-parameter form, while maintaining a good approximation for the service time. We demonstrate a tight fit of the estimates of the cost of communication based on the model with actual measurements of the communication and synchronization time between end processes. We compare the hyperbolic model with other two-parameter models and, in appropriate limits, show its compatibility with the LogP model.

Keywords: communication modeling, parallel processing, cluster computing, workstation networks  
*Submitted to Journal of Parallel and Distributed Computing*

---

<sup>\*</sup>Graduate Research Assistants, Computer Science Department, Old Dominion University, Norfolk, VA 23529-0162.

<sup>†</sup>Associate Professor, Computer Science Department, Old Dominion University, Norfolk, VA 23529-0162 and Senior Research Associate, Institute for Computer Applications in Science and Engineering, NASA Langley Res. Ctr., Hampton, VA 23681-0001. The work of this author was supported by the NSF under grant ECS-8957475 and by NASA under contract NAS1-19480 while the author was in residence at ICASE.

# 1 Introduction

The goal of this paper is to introduce a uniform framework for analyzing and predicting communication performance of parallel algorithms in real parallel processing environments. We include under “parallel processing environments” systems supporting computing both on traditional dedicated tightly coupled parallel computers (usually termed “multiprocessor systems”) and on clusters of loosely coupled workstations (usually termed “distributed systems”). However, “multitasking,” that is, the simultaneous execution of randomly interfering parallel jobs, is excluded.

There are two basic elements of a distributed parallel computation: the end processes that send, receive, manipulate and transform data and the links along which data flow, forming a network having both structural and dynamic properties. In designing and analyzing parallel algorithms, either we have to make assumptions about the properties of the software/hardware links over which messages are exchanged or these properties are implicit in the computational model used. The assumptions relate to the message reliability and the responsiveness of the communication network, the following being the most common:

(A<sub>1</sub>) Messages exchanged between end processes are not corrupted.

(A<sub>2</sub>) No duplicates of transmitted messages are generated.

(A<sub>3</sub>) Between any pair of end processes, messages are received in the order they were sent.

(A<sub>4</sub>) The delay is bounded, that is, it is guaranteed that a sent message will be delivered to the destination end process within a certain fixed time.

The overhead of enforcing these assumptions is sometimes not taken into account. Common idealizations include a communication delay equal to zero and unlimited bandwidth, independent of the number of processors. From the perspective of algorithm design, this approach provides a simple and tractable abstraction of a parallel computer. Early parallel models, beginning with [7], avoided the issue of communication and its impact on performance with the assumption of such “perfect” communication. Algorithms based on such models may appear to be highly performant, but more realistic assumptions [4, 9, 21] about the underlying communication system may reveal significant degradation of their behavior.

Our approach is also motivated by factors showing the increasing importance of communication in the area of parallel/distributed computing:

- The need for improving evaluation of complexity and efficiency of parallel algorithms.
- Technological trends. The increasing performance and memory capacity of the processing nodes in parallel computers and in workstation clusters [8] place heavier demands on the communication between nodes. Technological advances in the communication and network interface technologies come at a slower pace than those in (micro)processor performance and increased memory capacity. This has had and will continue to have the effect of making communication overhead a principal bottleneck for the overall performance of parallel algorithms.
- The revival of distributed computing. There is an economically driven shift toward using existing clusters of workstations in high performance distributed computing as an alternative to dedicated parallel computers. Over sixty publicly available systems for workstation collaboration are annotated in [20]. The communication links and the communication software being embedded in general purpose operating systems running on the processing nodes have distinct features that must be considered.

This paper introduces a new communication model for the evaluation of end-to-end communication costs in parallel processing environments. The computational tasks are accomplished by end processes that communicate using message passing. Messages are passed through communication blocks, whose parameters characterize the overall hardware and software links. The communication network itself is a communication block whose overall parameters are presumably unknown, but derivable for a given message pattern. For situations commonly encountered in real systems: passing messages from the same source over multiple communication blocks, processing incoming messages from the same source in parallel by distinct processors or by the same processor, and concurrent access of a single communication block by different message sources, we give rules for reducing the corresponding communication topology to a single equivalent communication block.

Although the model is expressed in terms of message-passing primitives, it has applicability to other communication paradigms commonly used in parallel programming. For example, the shared memory model of communication can be expressed in terms of a message passing model through the communication primitives send and receive.

Assumptions  $(A_1)$  -  $(A_4)$  above are related to the reliability of the communication network. It is the responsibility of the underlying layers of communication protocols (software links) to ensure that these assumptions will be always true for any end process or any pair of end processes participating in the computation. This is achieved in common operating systems by a layered communication architecture. In the case of tightly coupled parallel computers these properties can be supported directly by the hardware (through a highly reliable interconnection network and simple communication protocols). Although tightly coupled multiprocessing fits the hyperbolic model formalism, this paper is primarily motivated by the need to model communication within a cluster of workstations cooperating in a distributed computation. In the last section, we compare the hyperbolic model to the LogP model [4] in this context. Distributed computing is more challenging than multiprocessing for reasons beyond the obvious higher average latency and smaller average bandwidth per node:

1. The individual nodes in cluster computing environments are powerful full-function computers with a fully developed memory hierarchy, running a general purpose operating system.
2. Unlike a multiprocessor system, a network of workstations has no dedicated hardware links between processing nodes. In one form or another, depending on the physical and data link layer characteristics, contention does occur. Of course, contention might also occur in multiprocessor communication; however, its extent and impact can be limited by either the specific interconnection topologies or by a careful implementation of a given parallel algorithm.

An important consideration in evaluating the performance of parallel algorithms in practice, in multitasking environments where the processing nodes are not guaranteed to be available at all times, is the distribution of work among the processing nodes. This impacts more than the computation; communication may also be affected by the presence of other processes contending for the use of the communication links. However, it is not the goal of the present work to address the problem of load fluctuation and its possible effects on the performance of parallel algorithms either from the computation or from the communication point of view. We assume throughout the paper that a single end process is running at a given processing node. The opposite extreme (interfering end processes, but free communication) is explored, for example, in [13].

Evaluation of the validity of the model is limited in this paper to communication patterns common in scientific computations, on a cluster of dedicated workstations. We have obtained and will present in a companion paper experimental evaluations of the hyperbolic model with full prototype scientific applications, on both clusters of workstations and commercial MPPs. The reader is also referred to prior versions [18] and [19].

The paper is organized as follows:

Section 2 reviews related work that provides the context into which the hyperbolic model is introduced.

Section 3 formally defines the hyperbolic model and an algebra of four rules for reducing a communication graph to a single communication block. Each of the rules is illustrated with a simple example.

Section 4 describes communication patterns generated by operations common in parallel algorithms (broadcast, global reduction, synchronization, and nearest neighbor communication) and describes how the time required for these patterns can be evaluated using the model. Experimental results for these patterns in a distributed computing environment are presented to validate the model. Predictions and experiments disagree by at most 20% over a range of message sizes from one byte to 128 Kbytes, on up to 16 dedicated workstations connected by Ethernet.

Section 5 summarizes the LogP model of computation for massively parallel processors [4] and shows favorable comparison (agreement in predicted timings to within a factor of 3/4) between the hyperbolic and LogP models in the small message regime, where the basis for comparison is unambiguous.

## 2 Related Work

The need for realistic models of communication costs has been recognized for many years. Models accounting separately for startup and service times of messages are the most common, being based on an empirically inferred linear dependence of the time needed to send a message between two communicating parties on the size of the message. Various hardware and software overheads in a parallel environment that are modeled by a fixed component, independent of the message size, and by a variable component, proportional to the message size are identified, for example, in [6, 14, 15].

A general paradigm for architecture-independent performance evaluation has been proposed by Hockney and Jesshope [11], and is known in literature as the  $(r_\infty, n_{1/2})$  model, where  $r_\infty$  represents the asymptotic peak performance, and  $n_{1/2}$  the message length for which half of the asymptotic performance is achieved. Developed initially as a two-parameter model for the performance of vector computers as a function of the vector length, it can be used to characterize any process that has a linear timing relation with respect to some variable. Similar schemes were derived for modeling memory access overhead [10] or synchronization overhead [9]. This approach also provides a general framework for benchmarking supercomputers.

The bulk-synchronous parallel model (BSP) [21] is a model bridging software and hardware aspects of general purpose parallel computation. It is an abstraction of a parallel machine on which algorithms written in high-level languages are compiled with the goal of obtaining optimal memory access and communication schedules under the constraint of load balancing. The regularity of memory assignment and communication patterns in many computations motivates the concept of periodically synchronized phases of communication and computation. Besides the parameter quantifying this periodic pattern, the BSP model contains a parameter representing the bandwidth of an abstract router. Each communication phase is considered sufficiently voluminous that latency can be neglected.

The LogP model [4] is designed for point-to-point communication in multiprocessor systems. The model is characterized by four parameters: latency ( $L$ ), overhead ( $o$ ), reciprocal of bandwidth ( $g$ ), and the number of processors in the system ( $P$ ). These parameters are used to model fixed-sized short messages; long messages are assumed to be transmitted as a sequence of short messages. However, many modern parallel systems (e.g., the IBM SP series) provide direct hardware support for transmission of large messages. Since the LogP model does not recognize such support, it has

been recently extended by adding a new parameter: the reciprocal of bandwidth for long message ( $G$ ). The new model, called LogGP [1], has been applied to some common communication patterns, such as single-node scatter. These models are effective and accurate in predicting algorithm performances, though the large number of parameters makes the analysis complex. Moreover, adjustments representing additional architectural information are needed in particular applications (e.g., whether a processor can concurrently handle incoming and outgoing messages).

None of models mentioned, initially developed for richly interconnected parallel processors, accommodate contention in a general fashion. Schemes for partially avoiding contention in routing architectures (e.g., a hypercube in [21]) and for obtaining probabilistic guarantees for propagation times are proposed, but the problem of quantifying the effect of coexisting messages over the same link on the end-to-end communication performance requires more attention.

The hyperbolic model introduced in this paper is a variation on the two-parameter models. Its main goal is to address in a uniform way the modularity increasingly present in modern parallel computing environments, where a message path between two communicating parties crosses multiple processing modules having clearly defined interfaces and distinct functionality. If the twin parameters of every module on a message path are known (either by measurement or functional specification), the hyperbolic model allows them to be combined by a set of simple rules into a single pair of end-to-end parameters. In contrast to models that attempt to globally characterize communication costs independently of data paths, the modular hyperbolic representation is data-driven. It can thus take advantage of knowledge of connectivity and component parameters along the communication paths to adapt to specific patterns of communication. One benefit of its modularity and the application of the reduction rules to particular coexisting communication patterns is the accommodation of contention.

### 3 The Hyperbolic Communication Model

*Given a set of source nodes  $S$ , a set of destination nodes  $D$ , and a set of messages  $M$  in a parallel processing environment such that:*

1. *every message in  $M$  is sent by a node in  $S$  to a node in  $D$ ;*
2. *every node in  $S$  sends at least one message and all messages it sends are in  $M$ ;*
3. *every node in  $D$  receives at least one message and all messages it receives are in  $M$ ;*

*our goal is to estimate for every message in  $M$ :*

- *the time interval between the sending of the message and its delivery to the destination*<sup>1</sup>;
- *the time interval required by the source node to send it;*
- *the time interval required by the destination node to receive it.*

The sets  $D$ ,  $S$ ,  $M$  determine the state of the communication system, which is represented as a directed graph called a *communication graph* ( $CG$  for short). A  $CG$  has two types of nodes: terminal nodes and internal nodes. The terminal nodes represent the end processes that ultimately initiate the sending (source node) and receiving (destination node) of the data, while the internal nodes embed all the functions performed in software and hardware by the communication protocols

---

<sup>1</sup>More precisely, this is the time interval between the moment when the sender initiates the message transmission, and the moment when the last bit of the message is received at the destination.

in order to deliver data from source to the destination. The direction of graph edges specifies the data flow.

Between any two terminal nodes the data is passed in byte streams of any size called *messages*. A message is generated by only *one* source node and delivered to only *one* destination node. At the source node a message  $m$  is represented by a pair  $m(x, dest)$ , where  $x$  is the message size and  $dest$  is the destination node to which the message is to be delivered. At lower communication levels a message is usually split in smaller data units of limited size, called *packets* (if the message is small enough to fit in a packet then, obviously, it is not split). Associated with each edge is a list of the messages sent along that edge.

An internal node or *Communication Block* (*CB* for short) is an abstract module that performs the communication protocol functions. Among these functions are: splitting of messages in packets for passing to another *CB*, recovering lost or corrupted packets, and routing the packets in the network.

We say that two or more *CBs* are **dependent** iff only one of them can process data at a moment in time and **independent** iff at any moment in time all *CBs* can process *different* streams of data without interfering. For example, two *CBs* running on different processors are *independent*, while if they run on the same processor they are *dependent*.

The most important parameter characterizing a *CB* is the time required to process a message of size  $x$ , called the *total service time*. As with any realistic model, we consider that the packet processing time has two components [15]:

- a *fixed service time* that is independent of the packet size,
- an *incremental service time* that is proportional to the packet size.

The *fixed service time* appears at almost every layer of the communication architecture and includes [3, 14]: the overhead associated with memory management, interrupt processing and context switching, and the propagation delay of a packet on the communication network.

The *incremental service time* is mainly due to [3, 14, 15]: data movement between different protocol layers, building CRC (or checksum) when the packet is sent and verifying it when the packet is received, possible data compression/decompression, and transmission of the packet on the communication network.

For example, consider a distributed application consisting of three processes running on different workstations connected by a communication network. Assume that each workstation has a general purpose processor that runs the operating system and user applications, and a special I/O processor that sends/receives data to/from the communication network (the network adapter). The *CG* corresponding to this system is presented in Figure 1, where terminal nodes are represented by circles and *CBs* are represented by boxes. Each workstation is represented by two *CBs*, one that runs on the main processor (boxes labeled  $CB_{11}$ ,  $CB_{21}$ ,  $CB_{31}$ ) and represents the communication protocol functions performed by the operating system and the application (e.g., network layer and the upper layers of ISO/OSI standard), and the other that represents communication protocol functions performed by the network adapter ( $CB_{12}$ ,  $CB_{22}$ ,  $CB_{32}$ ). We also represent the communication network as a communication block, labeled  $CB_c$ , for which the fixed service time is the delay introduced by the communication network and the incremental service time is the average time required to process one byte of data passed to  $CB_c$  (called *transmission time*). The incremental service time includes the overhead generated by the protocol layers to ensure the reliability (e.g., acknowledgement packets).

Now, let us consider a *CB* characterized by the following parameters: the maximum packet size  $p$  (bytes), the fixed service time per packet  $a$  and the incremental service time per byte  $m$ . Then

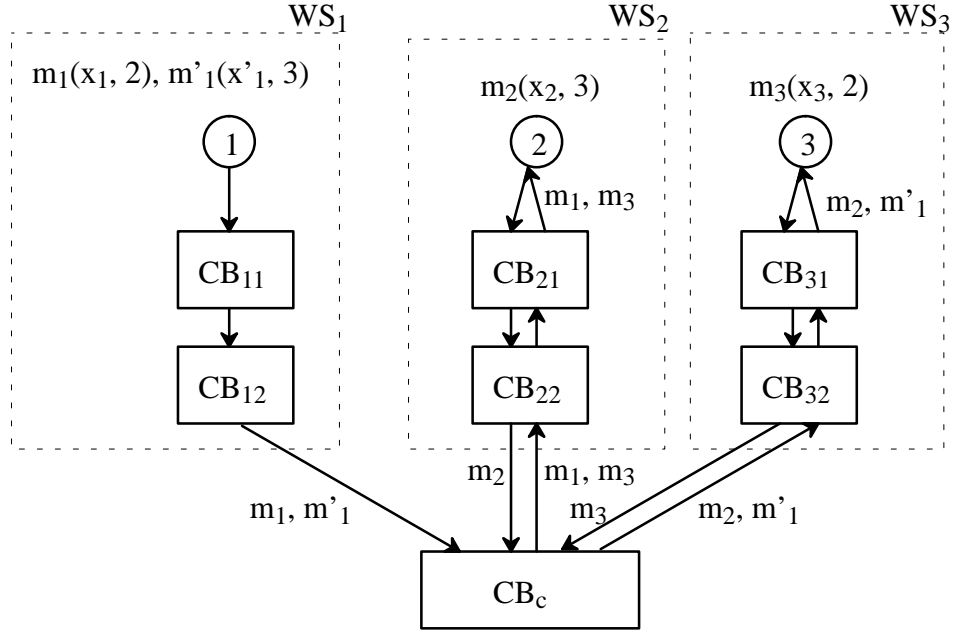


Figure 1: *Communication graph (CG) for three processes running on different workstations. Each terminal node (process) contains the list of messages it sends (node 1 sends  $m_1$  to 2 and  $m'_1$  to 3, node 2 sends  $m_2$  to 3, and node 3 sends  $m_3$  to 2). Each edge is labeled with the list of messages that flow in that direction.*

the total service time  $t$  for a message of size  $x$  is given by the following equation:

$$t(x; a, m, p) = a \lceil \frac{x}{p} \rceil + mx, \quad (1)$$

where  $\lceil x/p \rceil$  is the number of packets of maximum size  $p$  being processed. For convenience we rewrite Eq. (1) as:

$$t(x; a, m, p) = a \cdot \xi(x, p) + \left(\frac{a}{p} + m\right) \cdot x, \quad (2)$$

where  $\xi(x, p) = \lceil x/p \rceil - x/p = (p \lceil x/p \rceil - x)/p$  is a value between 0 and 1. Observe that for  $x \rightarrow 0$ ,  $t(x; a, m, p) \rightarrow a$  and for  $x \rightarrow \infty$  (i.e.,  $x \gg p$ ) the first term from (2) can be neglected, i.e.,  $t(x; a, m, p) \simeq (a/p + m) \cdot x$ . Using these observations, we approximate the total service time  $t$  with the following monotonically increasing continuous function defined on the interval  $[0, \infty)$ :

$$T(x; a, b) = \frac{a^2}{a + bx} + bx, \quad (3)$$

where  $b \equiv a/p + m$ . This is the equation of a hyperbola in the  $(x, t)$  plane, with a horizontal tangent and intercept  $a$  at  $x = 0$ , and an asymptote of slope  $b$ ; hence, the name of the model.<sup>2</sup>

Figure 2 shows both the hyperbolic fit  $T(x; a, b)$  and the best linear fit

$$l(x; a, b) = \frac{a}{2} + bx \quad (4)$$

<sup>2</sup>Notice that the parameter  $b$  in the denominator is not important for the correct behavior of the hyperbolic fit  $T(x; a, b)$  at the limits. Therefore, it could be used as a third parameter to improve the fit. However, since in practice it is difficult to evaluate it as an independent parameter, for convenience we choose it to be  $b$ .

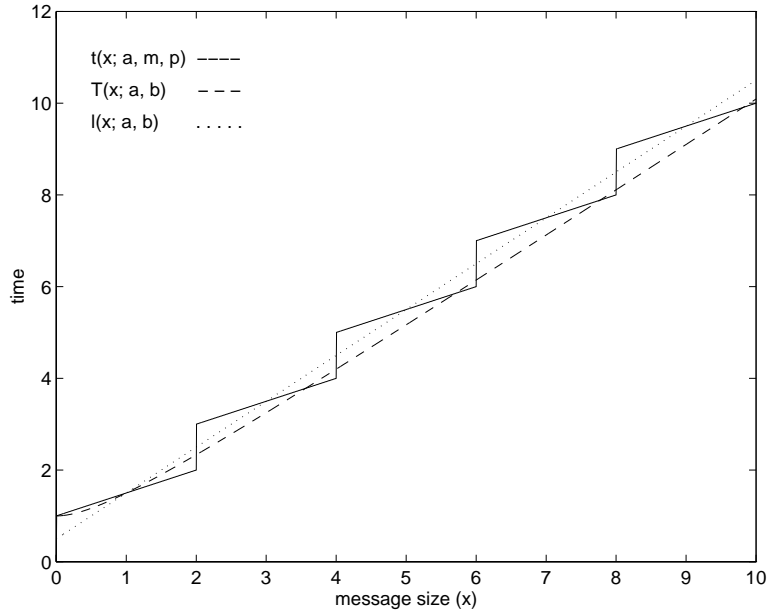


Figure 2: *The total service time  $t(x; a, m, p)$  versus the continuous hyperbolic and linear functions ( $T(x; a, b)$  and  $l(x; a, b)$ , respectively) used to approximate it ( $a = 1$ ,  $m = 0.5$  and  $p = 2$ ).*

for the service time given by the Eq. (1). While the hyperbolic fit better approximates small messages, the linear fit is more accurate for large messages. Specifically, when the message is very small ( $x \rightarrow 0$ ) the hyperbolic fit gives the exact value for the service time, i.e.,  $\lim_{x \rightarrow 0} T(x; a, b) = a$ , while the linear fit gives  $\lim_{x \rightarrow 0} l(x; a, b) = a/2$ . This is an important consideration in scientific computing, in which small messages, e.g., inner product reductions, occur frequently amidst other large messages. On the other hand, for large messages ( $x \rightarrow \infty$ ) the largest absolute error for the linear fit is  $a/2$ , while for the hyperbolic fit it approaches  $a$ . If we consider the worst case relative error, the hyperbolic fit outperforms the linear fit. In the worst case, the relative error for the linear fit is as large as 50% (for  $x \rightarrow 0$ ), while it is easy to see (Figure 2) that the worst case relative error for the hyperbolic fit is always less than 50%. It is possible to improve the accuracy of the linear model for small messages by replacing  $a/2$  with  $a$  as the zero-byte intercept in Eq. (4); however, in this case, the absolute error increases to  $a$ . Although this matches the maximum absolute error of the hyperbolic model, the maximum error occurs only asymptotically for the hyperbolic model, whereas it occurs throughout the entire message size range for the linear model. With either choice of intercept for the linear model, the average relative error of the hyperbolic model is better.

It is possible to improve the accuracy of the hyperbolic model for large messages by adding a third parameter. Namely, the service time could be given by:

$$T(x; a', b, c) = \frac{a'^2}{a' + bx} + bx + c, \quad (5)$$

where  $a = a' + c$ . Since the service time for large messages approaches the asymptote  $bx + c$ , the parameter  $c$  should be chosen such that to match the best linear fit for large messages (in our case the value of  $c$  would be  $a/2$ ). Also, notice that for  $a' = 0$ , Eq. (5) degenerates into the linear fit. Although the four combination rules presented in subsections 3.1 through 3.4 can easily be extended to include a third parameter, for simplicity we will present in this paper only the two-parameter model, in which the parameters  $(a, b)$  for a *CG* may be derived in terms of its elemental *CBs*. Using

$T_i$  to estimate the total service time required by  $CB_i$  (characterized by parameters  $a_i$  and  $b_i$  as  $CB_i(a_i, b_i)$ ) to process a message of a given size, we derive rules for reducing  $n$   $CB$ s interconnected in various structures to a single equivalent  $CB$ , with service time  $T(a_1, b_1, a_2, b_2, \dots, a_n, b_n)$ .

Next, we show the correspondence between the hyperbolic model and the parameters of the linear model proposed by Hockney and Jesshope [11]. This model is characterized by two parameters  $(r_\infty, n_{1/2})$ , where  $r_\infty$  represents the asymptotic bandwidth, and  $n_{1/2}$  represents the length of the message for which the performance decreases by half. Then the service time is a linear function of the length of the message [11]:

$$t(n; r_\infty, n_{1/2}) = \frac{n + n_{1/2}}{r_\infty}.$$

If we consider the best linear fit for the service time shown in Figure 2, then we obtain  $r_\infty = 1/b$  and  $n_{1/2} = a/(2b)$ . Alternatively, we can derive the parameter pair  $(r_\infty, n_{1/2})$  for the hyperbolic model, by using the service time expression given by Eq. (3):

$$r_\infty = \lim_{n \rightarrow \infty} \frac{n}{T(n; a, b)} = \frac{1}{b} \quad (6)$$

and

$$T(n_{1/2}; a, b) = \frac{2n_{1/2}}{r_\infty} \quad (7)$$

By solving Eq. (7), we obtain  $n_{1/2} = (\sqrt{5} - 1)a/(2b) = 0.618 a/b$ , which is within 25% from the value of  $n_{1/2}$  given by the linear model. Thus, when compared with the  $(r_\infty, n_{1/2})$  model, the hyperbolic model does not “lose” too much in approximating throughput of intermediate messages while better approximating the service time for small messages.

Finally, notice that although until now we have considered only a fixed and incremental service time per *packet*, the model can accommodate an additional fixed service time per *message*. This is useful in cases where the first packet of the message has a higher fixed service time than all subsequent packets. As an example, consider a network with wormhole routing; when the first packet of the message is sent a route is chosen between source and destination, and all subsequent packets of the same message are sent on the same route. Let us denote by  $a^{(1)}$  the fixed service time associated with the first packet and by  $a^{(2)}$  the fixed service time associated with all subsequent packets of the same message. The corresponding  $CB$  has the following parameters:

$$a = a^{(1)}; \quad b = \frac{a^{(2)}}{p} + m$$

where  $p$  is the packet size and  $m$  is the incremental service time per data unit. In this case the fixed service time associated with the message is  $a^{(1)} - a^{(2)}$ .

In the following four sections we present the basic interconnection schemes and derive the corresponding reduction rules.

### 3.1 Serial Interconnection

**Definition 1** *Two communication blocks  $CB_1(a_1, b_1)$  and  $CB_2(a_2, b_2)$  are serially interconnected with respect to a message  $m$  if every packet of message  $m$  is processed first by  $CB_1$  and next by  $CB_2$ , or first by  $CB_2$  and next by  $CB_1$ .*

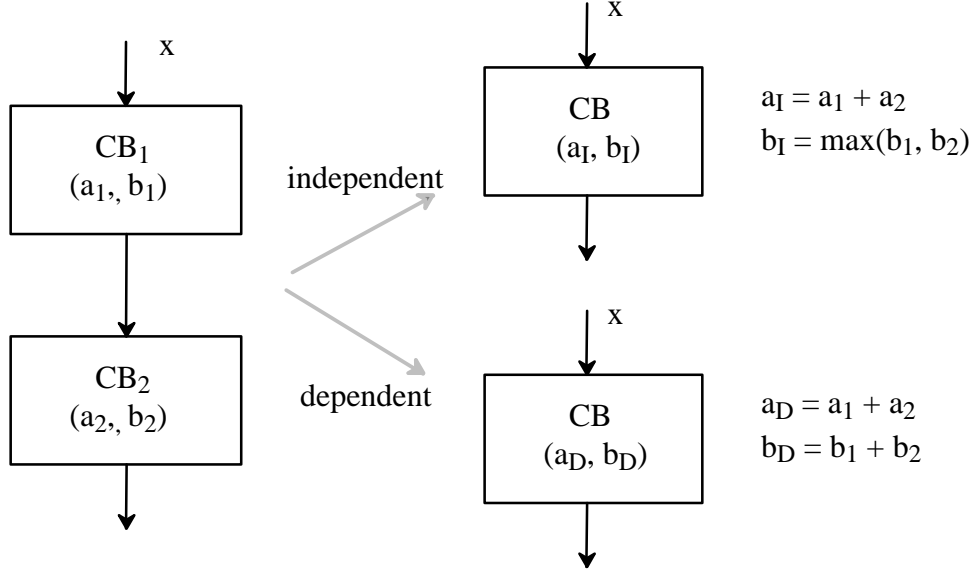


Figure 3: The equivalence transformation for two serially interconnected independent  $CB$ s (right top) and dependent  $CB$ s (right bottom).

Notice that this definition does not imply that a message is processed *in its entirety* by one  $CB$  and only after that by the other  $CB$ . In fact, if the message is long (greater than the maximum packet size) and the  $CB$ s are independent, as soon as  $CB_1$  delivers a packet,  $CB_2$  can start to process it. In other words (see Figure 3), while  $CB_2$  processes the packet most recently delivered by  $CB_1$ ,  $CB_1$  processes the next packet from its input message.

Next, we show how to transform this serial structure into an equivalent  $CB$  that has as input the input of  $CB_1$  and as output the output of  $CB_2$ . To determine the  $CB$  parameters we consider two cases:

1. *Independent  $CB$ s*. In this case,  $CB_1$  and  $CB_2$  run on different processors and therefore, as we have pointed out, they can concurrently process a long message. It is easy to see that when  $x \rightarrow \infty$  the dominant term in the total service time is  $\max(b_1x, b_2x)$ , due to the fact that either  $CB_1$  waits for  $CB_2$  to process the previous packet or  $CB_2$  waits for  $CB_1$  to deliver a new packet. On the other hand, when  $x \rightarrow 0$ , the whole message fits in a single packet and therefore  $CB_2$  cannot begin processing until  $CB_1$  finishes processing. Since the individual service times are  $a_1$  for  $CB_1$  and  $a_2$  for  $CB_2$ , it is clear that the total service time for  $CB$  is  $a_1 + a_2$ . Hence, we obtain the following parameters for the equivalent  $CB$ :

$$a_I = a_1 + a_2; \quad b_I = \max(b_1, b_2).$$

2. *Dependent  $CB$ s*. Here, it is not possible for  $CB_1$  and  $CB_2$  to run concurrently (i.e.,  $CB_1$  and  $CB_2$  use a non-sharable common resource during the processing). This is not different from previous case for  $x \rightarrow 0$  (the total service time is also  $a_1 + a_2$ ), but, since no processing overlap is possible, the total time service for long messages, i.e., when  $x \rightarrow \infty$ , becomes  $b_1x + b_2x$ . This gives us the following  $CB$  parameters:

$$a_D = a_1 + a_2; \quad b_D = b_1 + b_2.$$

Now, we can easily generalize our results by giving the following rule:

**Rule 1 (Serial Interconnection)** Given  $n$  serially interconnected communication blocks  $CB_i(a_i, b_i)$ ,  $1 \leq i \leq n$ , this structure is equivalent to a single communication block  $CB(a, b)$ , where:

$$a = \sum_{i=1}^n a_i; \quad b = \max(b_1, b_2, \dots, b_n) \quad (8)$$

if all CBs are independent, and

$$a = \sum_{i=1}^n a_i; \quad b = \sum_{i=1}^n b_i \quad (9)$$

if all CBs are dependent.

To illustrate the use of rule 1, consider a workstation modeled by three CBs :

- $CB_a(a_a, b_a)$  - models the total service time at the application level (e.g., suppose the application makes an extra copy to/from an internal buffer);
- $CB_{os}(a_{os}, b_{os})$  - models the total service time due to the communication protocol functions performed by the operating system;
- $CB_c(a_c, b_c)$  - models the total service time due to communication protocol functions performed by the network adapter. The  $a_c$  represents the time interval required to get access to the communication network (this is influenced by the medium access control mechanism [16]), while the  $b_c$  is the time required to send one data unit. The inverse of  $b_c$  corresponds to the available communication network bandwidth. The transmission delay (the time interval required to send one data unit from source to destination on the communication network) is ignored in this case as being much less than the other communication parameters.

As in the previous example, assume that the general purpose processor runs the operating system and the user processes, while the network adapter performs only specific communication network functions. We can reduce this structure by applying rule 1 twice: first reduce the serially interconnected *dependent* blocks  $CB_a$  and  $CB_{os}$  ( $CB_a$  and  $CB_{os}$  are dependent because they run on the same processor) to  $CB'$ , and next reduce the serially interconnected *independent* blocks  $CB'$  and  $CB_c$  to  $CB(a, b)$ . It is easy to verify that after these reductions we obtain the following CB parameters:  $a = a_a + a_{os} + a_c$ ,  $m = \max(b_a + b_{os}, b_c)$ .

### 3.2 Parallel Interconnection

**Definition 2** Two communication blocks  $CB_1$  and  $CB_2$  are parallel interconnected with respect to a message  $m$  if every packet of that message can be processed either by  $CB_1$  or  $CB_2$ .

Figure 4 shows two parallel interconnected communication blocks. For this type of interconnection we assume that the packets are processed in the way that minimizes the total service time of the message. As before, we take into account two cases:

1. *Independent CBs* . Let us denote by  $x$  the total size of the message  $m$ . According to our assumption, if  $x \rightarrow 0$  it is clear that the total service time is minimum when the input is entirely processed by  $C_1$  if  $a_1 < a_2$ , or by  $C_2$  otherwise. When  $x \rightarrow \infty$ , the total service time is minimized when the splitting of  $x$  ensures a equal load for both  $CB_1$  and  $CB_2$ . Denoting by  $x_1$  and  $x_2$  the

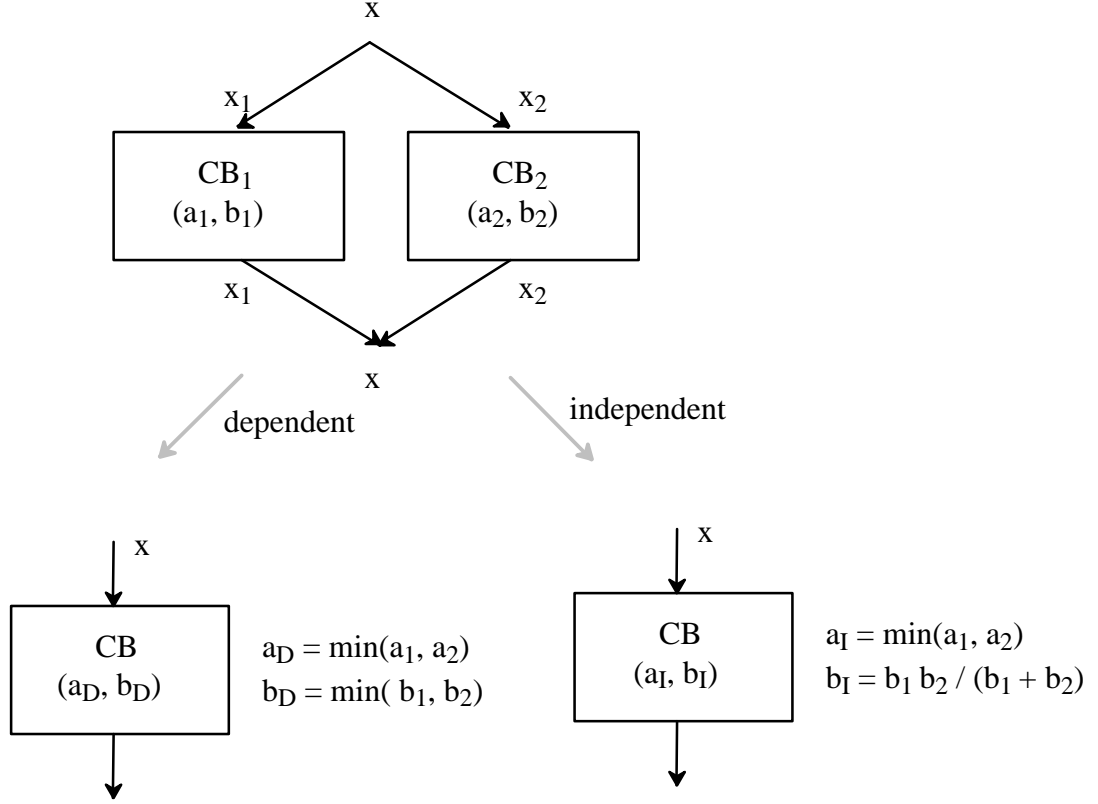


Figure 4: *The equivalence transformation for two parallel interconnected dependent CBs (bottom left) and independent CBs (bottom right).*

sizes of the inputs processed by  $CB_1$  and by  $CB_2$  respectively, it is easy to see that load balancing is achieved when  $x_1 = x b_2 / (b_1 + b_2)$ ,  $x_2 = x b_1 / (b_1 + b_2)$ . Finally, combining either of these solutions with the asymptotic expression of the total service time of  $CB$ ,  $T(x; a, b) = bx$  for  $x \rightarrow \infty$ , we obtain the overall  $CB$  parameter set:

$$a_I = \min(a_1, a_2); \quad b_I = \frac{b_1 b_2}{b_1 + b_2} \quad (10)$$

2. *Dependent CBs*. Since both  $CB$ s run on the same processor it is obvious that we can minimize the service time by simply choosing the best parameters in each case (e.g., for  $x \rightarrow 0$  we choose the  $CB$  that has the minimum fixed service time, while for  $x \rightarrow \infty$  we choose the  $CB$  that has the minimum incremental service time), which gives us the following results:

$$a_D = \min(a_1, a_2); \quad b_D = \min(b_1, b_2) \quad (11)$$

More generally, it can be shown that:

**Rule 2 (Parallel Interconnection)** *Given  $n$  parallel interconnected communication blocks  $CB_i(a_i, b_i)$ ,  $1 \leq i \leq n$ , this structure is equivalent to a single communication block  $CB(a, b)$  where:*

$$a = \min(a_1, a_2, \dots, a_n); \quad \frac{1}{b} = \sum_{i=1}^n \frac{1}{b_i} \quad (12)$$

if all *CBs* are independent, and

$$a = \min(a_1, a_2, \dots, a_n); \quad b = \min(b_1, b_2, \dots, b_n); \quad (13)$$

if all *CBs* are dependent.

Before turning to the more difficult case of concurrent message processing, we summarize the results of serial and parallel interconnection on independent and dependent *CBs*. In the small message limit that governs the  $a$  parameter, *CBs* in serial combine additively and *CBs* in parallel combine by taking the minimum. In the large message limit that governs the  $b$  parameter, *CBs* in serial that are dependent combine like resistors in series, and *CBs* in parallel that are independent combine like resistors in parallel. The other two subcases obey a maximum (serial, independent) or a minimum (parallel, dependent) law in deriving the overall  $b$ . No approximations are necessary in deriving these rules.

### 3.3 Concurrent Message Processing

Until now we have considered processing of individual messages of a given size. In this section we analyze the general case in which a *CB* receives  $n$  messages  $m_1, m_2, \dots, m_n$  of sizes  $x_1, x_2, \dots, x_n$  to be processed (Figure 5). We assume that *CB* processes  $m_1, \dots, m_n$  messages using an arbitrary policy, i.e., first processes  $m_{i_1}$ , next  $m_{i_2}$ , and last  $m_{i_n}$  (where  $i_1, \dots, i_n$  is a permutation of  $1, \dots, n$ ). Therefore, we cannot tell exactly how long it takes for *CB* to process a message  $m_i$  in the presence of other messages, but we know the corresponding total service time for each  $m_i$  if they are processed alone (3). Now let us consider that  $x_i \rightarrow 0$ , ( $i = 1, \dots, n$ ). In this case every message takes the same amount of time  $a$  to be processed and, therefore, that the total service time for all messages is  $na$ . Next, take  $x_i \rightarrow \infty$ . To compute the total service time we assume for simplicity that messages are processed sequentially without delays, and therefore the total service time is given by the following equation:

$$t(x_1, x_2, \dots, x_n; a, b) = b \cdot \sum_{i=1}^n x_i. \quad (14)$$

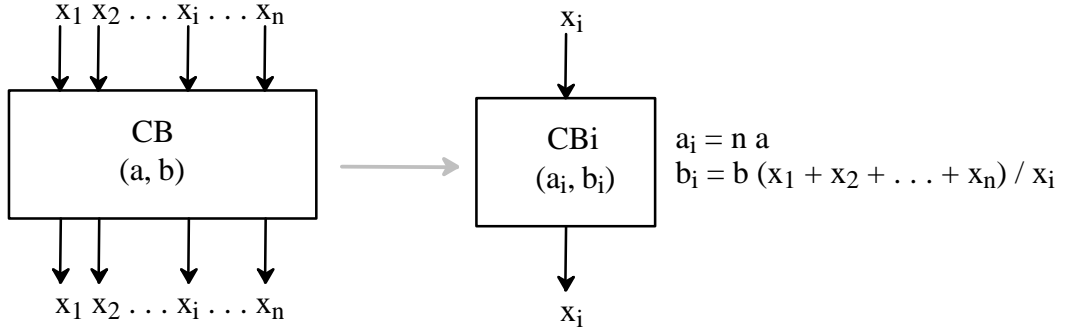


Figure 5: The equivalence transformation in the concurrent message processing case.

Since we cannot tell exactly when a particular message  $m_i$  is processed, we consider the time required to process  $m_i$  being bounded by the time required to process all messages, i.e., equivalent to the case in which  $m_i$  is the last message being processed. According to the previous limit conditions we can write the total service time as:

$$T(x_i|X, n; a, b) = \frac{(na)^2}{na + bX} + bX, \quad (15)$$

where  $X = \sum_{i=1}^n x_i$  is the total amount of information processed by  $CB$ . The “ $x_i|X, n$ ” notation indicates that the message of size  $x_i$  is processed concurrently with other  $n - 1$  messages of total size  $X - x_i$ . To be consistent, when  $X = x_i$  (which implies also  $n = 1$ ) we remove “ $|X, n$ ” from the notation. Then, with the notation  $a' = na$  and  $b' = bX/x_i$ , we write (15) as:

$$T(x_i; a, b) = \frac{a'^2}{a' + b'x_i} + b'x_i \quad (16)$$

By associating (16) with (3), we can state the third rule:

**Rule 3 (Concurrent Processing)** *A communication block  $CB(a, b)$  that processes  $n$  messages  $m_1, m_2, \dots, m_n$  of sizes  $x_1, x_2, \dots, x_n$ , respectively, is equivalent to a structure of  $n$  communication blocks  $CB_1(a_1, b_1), CB_2(a_2, b_2), \dots, CB_n(a_n, b_n)$ , where  $CB_i$  independently processes the message  $m_i$  and has parameters:*

$$a_i = na; \quad b_i = b \cdot \frac{\sum_{i=1}^n x_i}{x_i} \quad (17)$$

For the particular case in which all messages have the same length we obtain  $b' = bn$  (both parameters  $a$  and  $b$  are scaled with the same value  $n$ ). For a random order of messages,  $a_i$  is pessimistic by only a factor of two on average.

As an illustration of applying this rule (and of the first rule), we take a simple example. As depicted in Figure 6, suppose we have five processes (numbered from 1 to 5) running on different machines. Each machine is represented by a  $CB$  that includes all the communication protocol functions (implemented by the application, the operating system and on the network adapter). Further, assume that each of the processes 1 and 3 sends a message to process 4, while process 2 sends one message to processor 5. The question is to determine the total service time to send the message  $m_1$  from 1 to 4. To answer this question we reduce the initial  $CG$  (Figure 6(a)) in two steps. First, applying rule 3 to  $CB_c$  and  $CB_4$  we obtain an intermediate structure consisting of three serially interconnected independent communication blocks  $CB_1(a_1, b_1), CB'_c(a_c, b_c)$  and  $CB'_4(a'_4, b'_4)$  (Figure 6(b)), such that  $a'_c = 3a_c, b'_c = b_c(x_1 + x_2 + x_3)/x_1, a'_4 = 2a_4, b'_4 = b_4(x_1 + x_3)/x_1$ . Next, using rule 1 this structure is reduced to the final structure consisting of a single communication block  $CB$  (Figure 6(c)) with parameters  $a = a_1 + a'_c + a'_4$  and  $b = \max(b_1, b'_c, b'_4)$ , which are finally used to compute the total time to deliver  $m_1$  from process 1 to process 4 by substitution into Eq. (3).

### 3.4 The General Reduction Rule

Thus far, we have implicitly assumed that the communication graph is the same for small and large messages. Although this is true for many cases, for complex communication patterns this assumption is no longer valid. As an example, we will show that for the broadcast implementation (section 3.1) based on the binary tree topology for small messages we can ignore the contention on a shared communication network if the transmission time is orders of magnitude less than the sending and receiving overhead, while for large messages the contention cannot be ignored. Consequently, the  $CG$  will be different for small and large messages. In this case the following general reduction rule may be used:

**Rule 4 (General Reduction)** *Given two terminal nodes  $s$  and  $d$  such that  $s$  sends a message  $m$  of size  $x$  to  $d$ , then the total service time for the message  $m$  is:*

$$T(x; a, b) = \frac{a^2}{a + bx} + bx \quad (18)$$

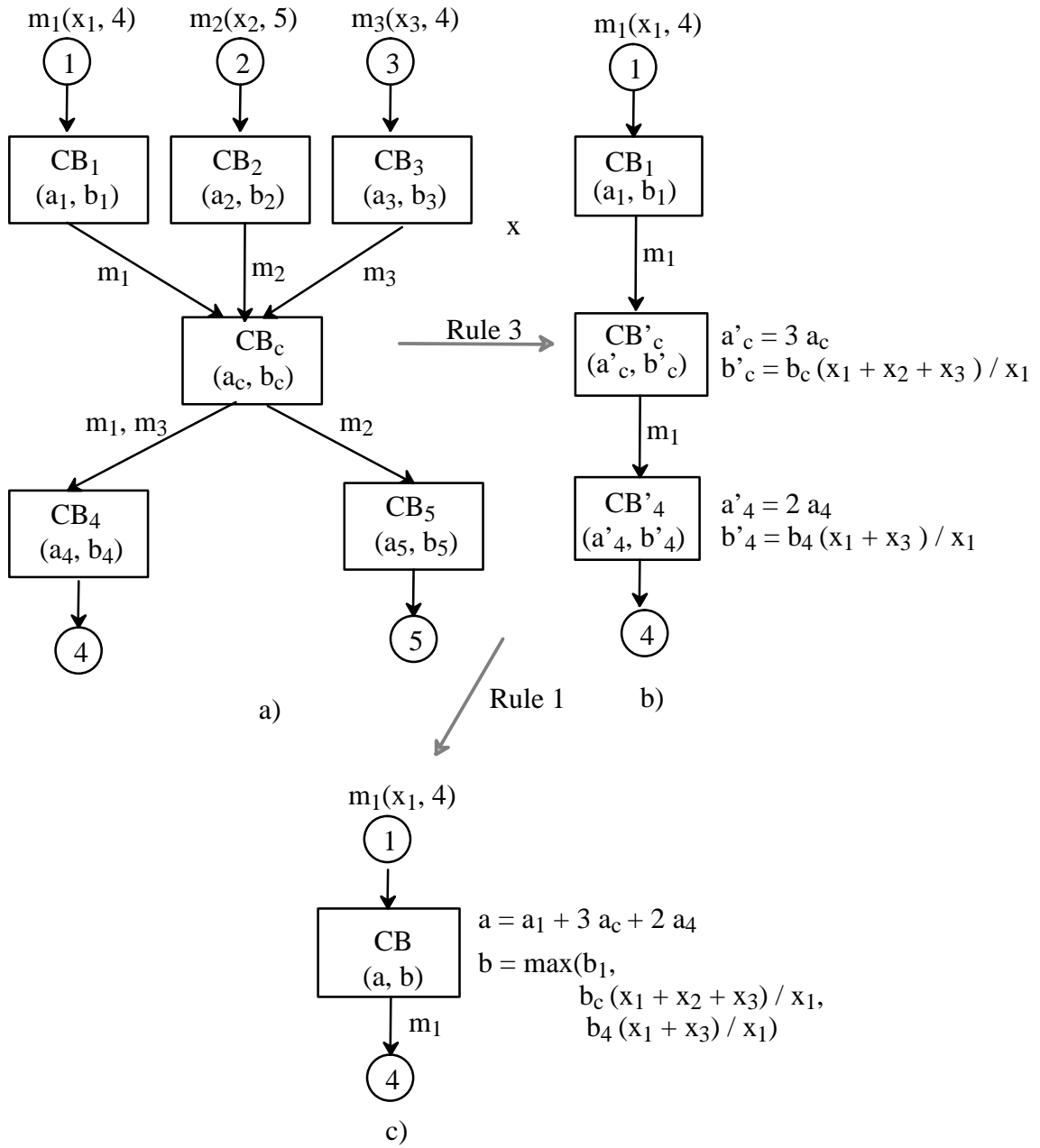


Figure 6: Computing the estimated time for the message  $m_1$  to be delivered to the process 4 by successive reductions of the communication graph.

where  $a$  is the service time when sending a small message from  $s$  to  $d$  ( $x \rightarrow 0$ ), and  $b$  is the service time per data unit when sending a large message from  $s$  to  $d$  ( $x \rightarrow \infty$ ).

The parameters  $a$  and  $b$  can be computed by using rules 1-3 to reduce the corresponding  $CG$ s. Notice that the general reduction rule is equivalent to reducing the paths along which message  $m$  may travel between source and destination (called  $m$ -communication paths) to a single communication block  $CB_G$  with parameters  $a$  and  $b$ .

### 3.5 Communication Time Measures

When a message is sent between two end processes, represented as terminal nodes in  $CG$ , three measures are particularly important:

- the total time interval between sending the message (by the source process) and delivering it (to the destination process), called *total communication time* ( $T_c$ ). As we have shown in the previous section, by applying the general reduction rule,  $T_c$  can be computed as the *total service time* of the resulting  $CB_G$ .
- the time spent by sender while sending the message, called *sending time* ( $T_s$ ).
- the time spent by receiver while receiving the message, called *receiving time* ( $T_r$ ).

To determine the  $T_s$  and  $T_r$ , we need to take a closer look at the sending and receiving mechanisms. First, let us consider all paths between source and destination along which a message travels. Next, using the equivalence transformation rules, we reduce all paths to a single path containing only independent  $CB$ s :  $CB_1, CB_2, \dots, CB_n$ , where the source process runs on the same processor as  $CB_1$  and the destination process runs on the same processor as  $CB_n$  (we consider that this is always possible). Now, let us analyze the mechanism of sending a message from source to destination along the equivalent path. The discussion here is similar to the serial interconnection of independent  $CB$ s . If the message is large, i.e., it consists of a large number of packets, then the message is concurrently processed by the independent  $CB$ s in a pipeline fashion. As we have shown, in this case the processing speed is determined by the slowest  $CB$ . From here results that if  $CB_1$  is not the slowest communication block, then after it processes a packet it must wait a certain amount of time in order to deliver the next packet to  $CB_2$ .

We define  $T_s$  either as the time required to process all message packets by  $CB_1$ , or as the time interval between starting processing of the first packet and the delivery of the last packet to  $CB_2$ . In the first case, the send primitive is said to be *preemptive*, while in the latter the send primitive is said to be *non-preemptive*. When a *preemptive* send primitive is used, the control is returned to the caller process as soon as the send operation is initiated and further computation can be performed concurrently with the processing required to send the message. When a *non-preemptive* send primitive is used the caller process is blocked from the moment of calling the send primitive until the last packet of the message is delivered to  $CB_2$ . Since our main focus is to determine the real processing time spent by a  $CB$  in sending/receiving a particular message, we prefer to use the terms *preemptive* and *non-preemptive* to characterize the communication primitives, rather than overloaded terms, such as *blocking/non-blocking* and *synchronous/asynchronous*, which are often used to define the semantics of the communication primitives. Differences between various types of communication primitives (see [5] for an extensive discussion and formal treatment) are ultimately captured in the  $CB$  parameters. What is important from the point of view of performance evaluation is the extent to which concurrent processing by the application process and its neighboring  $CB$  is allowed.

As an example of a *preemptive* send primitive, let us consider a single processor workstation that runs a preemptive operating system (e.g., UNIX). We roughly describe how the send primitive may be implemented. When the application process (that runs on the same processor as the operating system) invokes the send primitive, the first packet of the message is processed and delivered to the  $CB_2$ . Next, the control is returned to the caller process, which can proceed with its computations. After  $CB_2$  processes and delivers the current packet, it asks  $CB_1$  for the next packet to be sent (usually, this is done using an interrupt mechanism). In turn, the application process is interrupted and the next packet is processed and delivered to  $CB_2$ . This procedure continues until the last packet of the message is sent out. If we neglect the interrupts and operating system call overhead, then  $T_s$  is the total time required by  $CB_1$  to process all the packets of the message.

In the case of the *non-preemptive* send primitive implementation, after the first packet of the message is processed and delivered,  $CB_1$  waits to deliver the next one. Therefore the sender process is blocked until the last packet of the message is delivered to  $CB_2$ .

To determine  $T_s$  we consider several cases (see Table I):

- if the message is small, i.e., it fits in one packet, we take  $T_s$  equal to  $CB_1$  service time  $T_{CB_1}$  for both *preemptive* and *non-preemptive* send primitives. This is equivalent to considering that when the send primitive is invoked, the message is processed and delivered in only one packet to  $CB_2$  and then the control is returned to the application process.
- if the message is large and a *non-preemptive* send primitive is used, then the total communication time  $T_c$  is equal to  $T_s$  plus the time required by the last message packet to be delivered to the destination process, i.e.,  $T_{CB_n}$ . Therefore we can take as an upper bound for  $T_s$  the total communication time  $T_c$ .
- if the message is large and a *preemptive* send primitive is used, then  $T_s$  accounts for the total time required to process and deliver all the packets of the message by  $CB_1$ , and thus we take  $T_s$  equal to  $T_{CB_1}$ .

Although we have considered very simple send primitive implementations, the model can accommodate more complicated implementations. As an example, let us assume that the communication protocol requires that the receiver be informed about the size of the message before the message is actually sent (in order for the receiver to allocate memory space for the incoming message). Moreover, consider that this implementation is based on exchanging two messages: one to inform the receiver about the size of the message and one to acknowledge that the buffer has been allocated and the sender can proceed. This case can be modeled by adding a new independent communication block before  $CB_1$ , called  $CB_0$ , with the following parameters:  $a$ , equal to the average time required to exchange the two messages plus the overhead to allocate the memory at the receiver and possibly other interrupt and system calls overheads, and  $b = 0$ .

As another example, let us assume that for a *non-preemptive* send primitive implementation the communication protocol requires that every packet be acknowledged by the receiver. In this case we can add a new communication block  $CB'_1$  after  $CB_0$ , which has parameter  $a$  equal to the average time for receipt of the acknowledgement, and  $b = 0$ .

Now, let us concentrate on the receiving time  $T_r$ . Since we are not interested here in the synchronization time, we consider that the receive primitive is called at the same time the first packet of the message is received by  $CB_n$ . Similarly to  $T_s$ ,  $T_r$  is defined either as the time required to process all packets of a message by  $CB_n$  (*preemptive* receive primitive), or as the time interval between the beginning of processing of the first packet and the finishing of processing of the last packet from the message (*non-preemptive* receive primitive). In general, the  $T_r$  analysis is the

Table I: *Sending ( $T_s$ ) and receiving time ( $T_r$ ) expressions, where message size is  $x$ .*

$T_s$ :	preemptive	non-preemptive
short message ( $x \rightarrow 0$ )	$T_{CB_1}(x; a_1, b_1)$	$T_{CB_1}(x; a_1, b_1)$
long message ( $x \rightarrow \infty$ )	$T_{CB_1}(x; a_1, b_1)$	$T_c(x)$

$T_r$ :	preemptive	non-preemptive
short message ( $x \rightarrow 0$ )	$T_{CB_n}(x; a_n, b_n)$	$T_{CB_n}(x; a_n, b_n)$
long message ( $x \rightarrow \infty$ )	$T_{CB_n}(x; a_n, b_n)$	$T_{CB_n}(x; a_n, b_n) \leq T_r \leq T_c(x)$

same as  $T_s$  analysis (see Table I). The only difference is when we consider large messages and *non-preemptive* receive primitives. Here, we take two cases. First, if we assume that the application receives an isolated message, then we take  $T_r$  equal to  $T_c$  (since the message is not passed to the application process until its last packet is received). On the other hand, if more than one message is received at the same time, the waiting time between processing packets from the same message can be used to process packets from other messages, and therefore, in the limit, we can take  $T_r$  equal to  $T_{CB_n}$ .

Although expressions for  $T_s$  and  $T_r$  are given only for extreme message sizes ( $x \rightarrow 0$ ,  $x \rightarrow \infty$ ) in Table I, we can again use Eq. (3) to approximate  $T_s(x; a_s, b_s)$  and  $T_r(x; a_r, b_r)$  for any message size  $x$ , where  $a_s = T_s(x \rightarrow 0)$ ,  $a_r = T_r(x \rightarrow 0)$  and  $b_s = \lim_{x \rightarrow \infty} \frac{T_s(x)}{x}$ ,  $b_r = \lim_{x \rightarrow \infty} \frac{T_r(x)}{x}$ .

## 4 Common Communication Patterns in Parallel Applications

In this section we illustrate how the model can be used to analyze four archetypal communication patterns encountered in parallel applications: broadcast, synchronization, global reduction, and nearest neighbor communication.

We consider a network of homogeneous workstations interconnected by a communication network. Each workstation is represented by a communication block  $CB_W$ , while the communication network is represented by a communication block  $CB_C$ . Also, when a message is received, a communication block  $CB_L$  is added between the communication network  $CB_C$  and the receiver  $CB_W$ . The role of  $CB_W$  is to capture the message processing overheads at send and receive (here we assume that the send and receive processing overheads are equal).  $CB_C$  captures the communication network bandwidth ( $1/b_C$ ) and the possible delay before the first bit of the packet is sent on the network ( $a_c$ ). Finally,  $CB_L$  captures the communication delay  $L$  ( $a_L = L$ ,  $b_L = 0$ ). The send and receive primitives are considered *non-preemptive*. Figure 7 shows the communication graph for a message transmission between two processors.

The values of all the  $CB$  parameters, excepting  $a_L$ , were experimentally determined in the limits of small or large message size and two or many processors as described in [19]. The  $a_L$  parameter was taken to be the maximum delay on an Ethernet LAN [17].<sup>3</sup> All experiments in this section were run during periods of dedicated time on up to 16 Sun SparcstationELC workstations. The p4 package from Argonne National Laboratory [2] served as the application-level communication support.

<sup>3</sup>It is worth noting that in this case  $a_L$  could be as well neglected since in an Ethernet network its value is up to three orders of magnitude less than that of  $a_W$ . However, we have chosen to keep it in order to show how the final parameters are computed in a more general case.

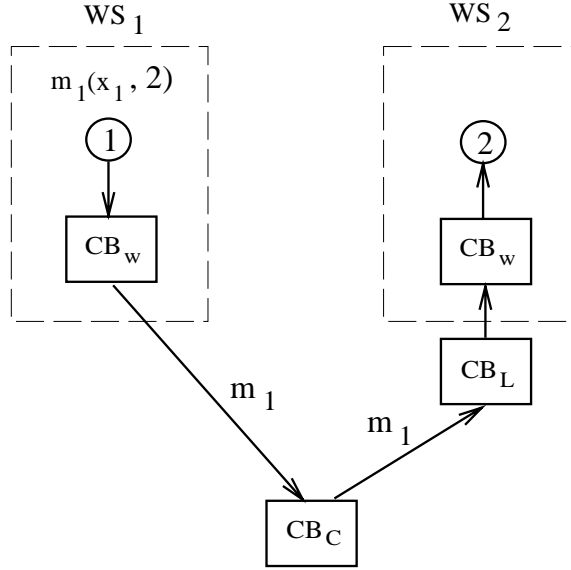


Figure 7: *The communication graph for sending one message from process 1 to process 2. Observe that the  $CB_L$  appears in the communication path only between communication network ( $CB_C$ ) and workstation communication block  $CB_W$ .*

#### 4.1 Broadcast

The broadcast primitive ensures the delivery of a message from one processor to  $N$  other processors. We consider two broadcast implementations. First, a binary tree is used to broadcast the message from a root processor to all other processors as indicated in Figure 8. For simplicity, we assume that every node of index  $i$  sends the message first to the left child ( $2i + 1$ ) and next to the right child ( $2i + 2$ ). We are interested in determining the total time required to complete the broadcast, i.e., from the moment when the root begins the transmission of the first message to the moment when the message is received by the last processor. As usual, two extreme cases are considered: the message is very small ( $x \rightarrow 0$ ), and the message is very large ( $x \rightarrow \infty$ ). For small messages we assume that sending and receiving overheads  $a_W$  are much larger than the actual transmission time  $a_C + b_C x$  and therefore we do not address the situations in which more than one processor sends the message on the communication network at the same time. With this assumption, the communication time between any two processors is  $T_c = 2a_W + a_C + a_L$ , while the sending and receiving times are  $T_s = T_r = a_W$ . With reference to the particular incomplete binary tree example in Figure 8(b)), the time required to complete the broadcast is  $8a_W + 3a_C + 3a_L$ . Generally, for a complete binary tree of height  $h$ , the time to complete the broadcast is  $h(3a_W + a_C + a_L)$ .

In the case of large messages the transmission time and other incremental service times are much larger than the communication delay and corresponding fixed service times. The activity of each processor over time is depicted in Figure 8(c). Since we consider *non-preemptive* send and receive primitives, we have  $T_s = T_r = T_c$  (see Table I). As shown, there are moments in time when more than one processor sends a message on the communication network (e.g., transmission between processor 0 and 2 takes place simultaneously with the transmission between processor 1 and 3). If there are  $n$  processors that concurrently send messages of the same size, then by applying rule 3 and the general reduction rule we obtain for every message path the equivalent communication block  $CB_G$  with parameters:  $a_G = 2a_W + na_C + a_L$  and  $b_G = \max(nb_C, b_W)$ . Since, for very large messages, the incremental service time dominates the fixed service time we approximate  $T_c$  with  $b_G x$ ,

where  $x$  is the size of the message being broadcast. Consequently, the time required to complete the example broadcast is  $x(2 \max(b_C, b_W) + \max(2b_C, b_W) + 2 \max(3b_C, b_W))$ . Now, the entire broadcast communication graph can be reduced to one communication block  $CB_{BCAST}$  with the following parameters:  $a_{BCAST} = 8a_W + 3a_C + 3a_L$ ,  $b_{BCAST} = 2 \max(b_C, b_W) + \max(2b_C, b_W) + 2 \max(3b_C, b_W)$ .

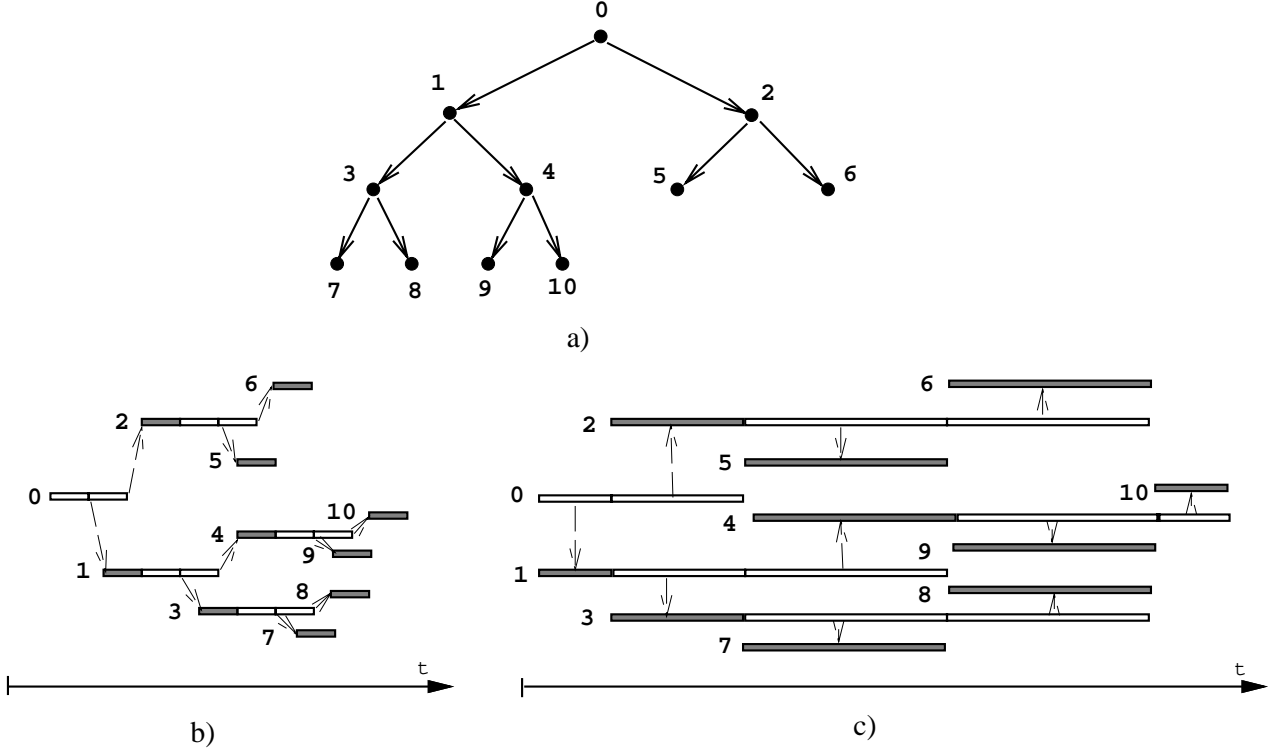


Figure 8: a) The broadcast binary tree for 11 processors. b) The processor activity when a small message is broadcast. The sending time is represented by empty bars while the receiving time is represented by shadowed bars. c) The processor activity when large messages are broadcast.

In the second broadcast implementation (which is the native implementation in p4 [2], version 1.3) the root processor simply sends the message to every other processor: 1, 2, ...,  $N$ . In this case the time to complete the broadcast is  $(N + 1)a_W + a_C + a_L$  for small messages and  $N \cdot \max(b_C, b_W) \cdot x$  for large messages. Although this implementation is the simplest possible, notice that if  $b_C > b_W$ , there is no other broadcast implementation to give better performance for large messages. (It is easy to verify that the binary-tree broadcast implementation is no better.) In this case the communication network is the bottleneck for any number  $i$  of messages that are concurrently sent ( $\max(ib_C, b_W) = ib_C$ ), and therefore the total broadcast time has as a lower bound the time required to send all messages across the communication network, which is  $Nb_C \cdot x$ .

Curves of the estimated communication time functions for the tree-based and serial broadcasts, together with experimental measurements of  $T_c$  are shown in Figures 9 and 10, respectively. The root mean square errors are 0.16 for the binary tree broadcast implementation and 0.14 for the p4 native broadcast implementation. In both cases, the maximum relative error was less than 20%. The model offers an accurate approximation to the actual measurements.

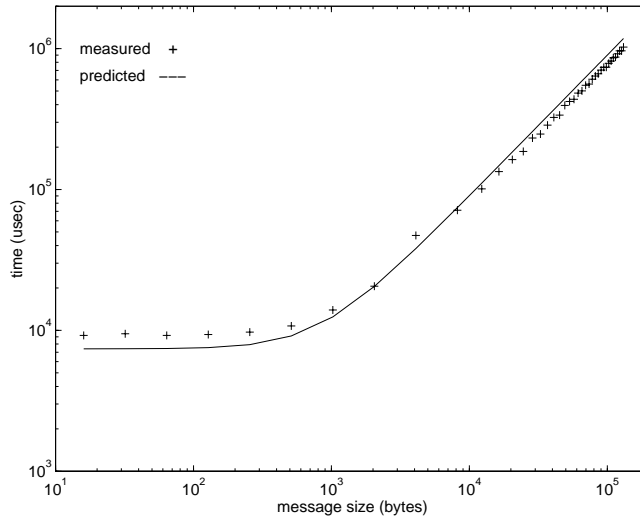


Figure 9: *The estimated  $T_c$  versus experimental data for the broadcast binary tree implementation. The root mean square error is 0.16. The experiments were run on 11 Sun SparcstationELC workstations interconnected by an Ethernet network using p4.*

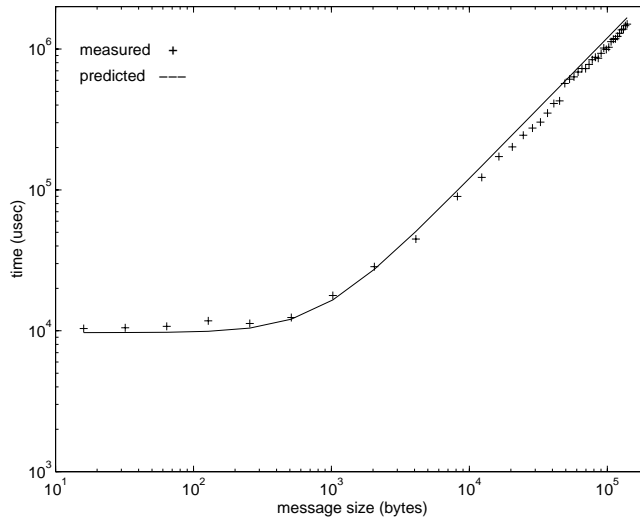


Figure 10: *The estimated  $T_c$  versus experimental data for p4 native broadcast implementation. The root mean square error is 0.14.*

## 4.2 Synchronization and Global Operations

Both synchronization and global operation primitives can be implemented using the same communication pattern. Although it is not the most efficient implementation, we describe here the one used in p4, version 1.3. The global operation implementation differs from the synchronization in two respects. First, during the global operation the synchronization messages carry partial results and second, besides sending and receiving messages, the processors are responsible for computing partial and final results. Therefore, synchronization can be seen as a special case of global operation where no computation is performed. In the remainder of this subsection, we concentrate on the global operation implementation.

A global operation primitive implements a group computation. Formally, a group computation is defined as follows: given  $n$  different items  $a_1, a_2, \dots, a_n$  in a group  $(S, \oplus)$  (where  $\oplus$  is a binary associative and commutative operation defined on set  $S$ ) compute the final value  $a_1 \oplus a_2 \oplus \dots \oplus a_n$ . Finding the sum, the maximum, or the minimum of a set of  $n$  numbers are examples of group computation.

The global reduction primitive gathers a value (or a set of values) from each processor, computes from them a single result (or a single set of results) and distributes it to every node. The implementation consists of two phases, illustrated by the light and dark arrows in Figure 11(a). In the first phase, the tree is used to collect the results from the leaves toward the root. Whenever a node receives the values from its children, it computes the partial result, i.e.,  $val_{node} \oplus val_{lchild} \oplus val_{rchild}$ , and sends it to the parent. Therefore, after the root receives the partial results from its children, it can compute the final result. In the second phase, the root distributes the final result by sending a message to every processor.

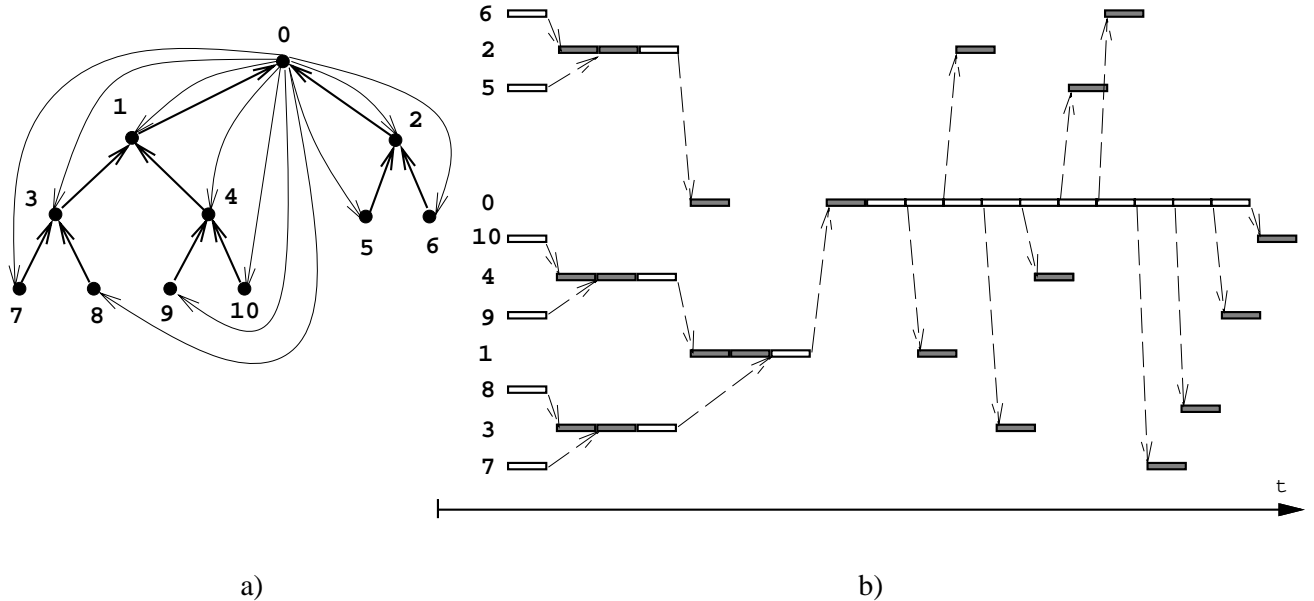


Figure 11: a) The communication pattern used by the synchronization and global operation primitives for 11 processors. b) The processor activity over time.

Since the values carried by the messages are often no larger than 8 bytes (double precision scalars), we assume that sending and receiving overheads are much larger than the actual data transmission time, and therefore we ignore the message contention on the communication network. Also, we ignore the time to compute the partial and final results as being much less than the

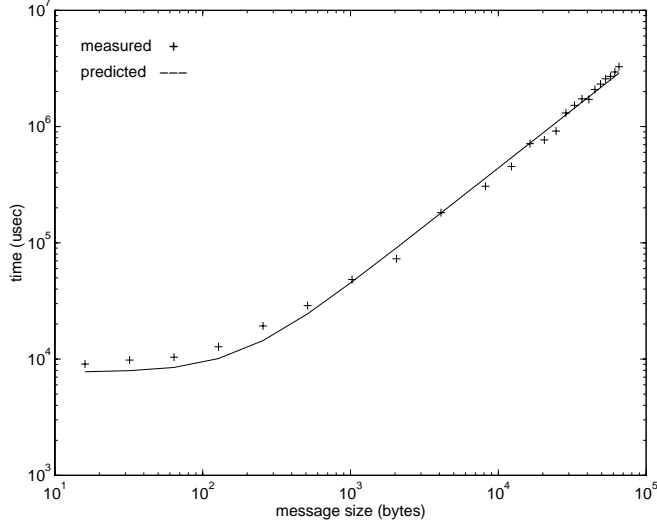


Figure 12: *The estimated  $T_c$  versus experimental data for neighbor communication pattern. The root mean square error is 0.12. Here,  $N = 10$  and  $K = 4$ .*

communication time. From Figure 11(b) it may be seen that the total time required to complete the global operation is  $19a_W + 4a_C + 4a_L$ . The observed relative error between the estimated completion time and experimental measurements is about 15%. (Since synchronization and global reduction operate on messages of trivial size, there is no effective hyperbolic law as a function of message size to graph for these primitives.)

### 4.3 Neighbor Communication

A broad range of scientific algorithms arising from differential equations require data to be sent from one processor to its logical neighbors. As an example, consider a domain decomposition problem [12] where each subdomain of a domain on which a partial differential equation is to be solved is mapped onto a single processor. At each iteration of the algorithm every processor sends to its neighbors the boundary data to be used in the next iteration.

More generally, suppose there are  $N$  processors and each of them has  $K$  ( $K < N$ ) logical neighbors. Further, we assume that every processor sends messages of the same length to each of its neighbors at the same moment of time. The latter assumption presumes a balanced parallel application in which every processor has the same amount of work to perform between synchronous phases of sending and receiving boundary data.

By applying the reduction rules 1 and 3 to the resulting  $CG$  it is easy to verify that the equivalent  $CB$  for any message path has the following parameters:  $a_{NEIGHBOR} = 2Ka_W + KNa_C + a_L$  and  $b_{NEIGHBOR} = \max(KNb_C, b_W)$ . Figure 12 shows the estimated  $T_c$  function versus experimental measurements with a least root mean square error of 0.12, and a maximum relative error of 16%.

## 5 The LogP Model

The LogP model [4] is designed for point-to-point communication in multiprocessor machines. The underlying architecture consists of modules connected by a communication network. A module contains a processor, a local memory and a network interface. The model assumes that send and

receive operations are performed by the main processor, i.e., there is not a specialized processor to perform network interface functions. This means that during the send or receive operations the main processor does not perform any other computation. The basic version of the LogP model assumes that all messages have the *same* size and that this size is *small*. The model is characterized by four main parameters:

1.  $L$ (latency) - the upper bound for the delay of a message transmission between the source and destination processors.
2.  $o$ (overhead) - the time interval required to send or receive a message. During this time the processor cannot perform any other operations.
3.  $g$ (gap) - the minimum time interval between two consecutive message transmissions or receptions.
4.  $P$  - the number of modules.

A fifth parameter to capture the enhanced bandwidth of long messages is incorporated in [1].

When a small message is sent, according to the LogP model, the communication time is equal to the sending overhead  $o$ , plus the delay time  $L$ , plus the receiving overhead  $o$ , i.e.,  $2o + L$ . On the other hand, when more than one message is sent by the same processor, a new message cannot be issued earlier than  $\max(g, o)$  and, therefore, the communication time to send  $n$  consecutive messages is  $(n - 1) \max(g, o) + 2o + L$ . The first term accounts for sending the first  $n - 1$  messages, while the last two account for sending the last message (see Figure 13). Since the second expression reduces to the first for  $n = 1$ , we consider further only the second.

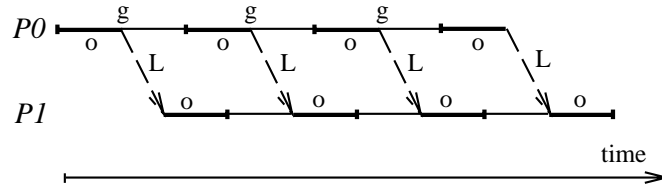


Figure 13: *The timing diagram for sending 4 consecutive messages, from P0 to P1, in the LogP model. Here,  $g \geq o$ .*

To capture the LogP parameters in the hyperbolic model we use the communication graph from Figure 7, where  $a_W = b_W = o$ ,  $a_L = L$ ,  $b_L = 0$  and  $a_c = 0$ ,  $b_c = g$ . Since the LogP model assumes that all messages are of a small fixed size, these will be interpreted as packets in the hyperbolic model, while consecutive messages sent by one processor (in LogP) will be interpreted as packets of a single message. Also, because all packets have the same size, for convenience, we assume a packet to be of unit length (i.e., each packet contains exactly one data unit). By applying Rule 1 to the communication graph, it is easy to see that the equivalent communication block has the following parameters:  $a = a_W + a_c + a_L + a_W = 2o + L$  and  $b = \max(b_W, b_c, b_L) = \max(g, o)$ . The total service time is given by (18).

When we write  $x \rightarrow 0$  in the hyperbolic model, we are referring to the smallest possible size of a message that can be sent, which can generally be much smaller than the packet size. However, in this case a packet consists of exactly one data unit (corresponding to a message in LogP), and therefore a message cannot be of a size smaller than a packet size. To accommodate this restriction within the formalism of the hyperbolic model, we take  $x = n - 1$ , where  $n$  is the size of the message. Thus,  $x \rightarrow 0$  in the hyperbolic model and  $n = 1$  in the LogP model refer to the identical limit,

namely that of the smallest message that can be sent. Next, if we denote by  $T_{hyp}(n)$  ( $= T(n-1; a, b)$ ) the communication time to send a message of size  $n$  in the hyperbolic model and by  $T_{LogP}(n)$  the communication time to send  $n$  consecutive messages in LogP model, we obtain:

$$T_{hyp}(n) = \frac{a^2}{a + (n-1)b} + (n-1)b; \quad T_{LogP}(n) = a + (n-1)b.$$

To see how much the estimated communication times for both models may differ, we consider the ratio  $T_{hyp}(n)/T_{LogP}(n)$ :

$$\frac{T_{hyp}(n)}{T_{LogP}(n)} = \frac{a^2 + (n-1)ab + (n-1)^2b^2}{a^2 + 2(n-1)ab + (n-1)^2b^2}.$$

It is easy to verify that for any value of  $n \geq 1$  and nonnegative  $a$  and  $b$ , we have:

$$\frac{3}{4} \leq \frac{T_{hyp}(n)}{T_{LogP}(n)} \leq 1. \quad (19)$$

Further, let us compute the sending (resp., receiving) time, i.e., the actual time required by a processor to send (resp., receive)  $n$  consecutive messages, for both models. For the LogP model, clearly, we have (see Figure 13)  $T_{s\_LogP}(n) = T_{r\_LogP}(n) = no$ . Next, notice that if  $g > o$ , after a message is sent, the processor is free for time  $g - o$  to perform other computations. Since we have interpreted consecutive messages sent by the same processor in the LogP model as packets of a single message in the hyperbolic model, between any two consecutive packets sent or received in the hyperbolic model, the processor can perform other computations. Therefore, the equivalent send and receive primitives of the hyperbolic model are *preemptive*. From Table I we thus have:

$$T_{s\_hyp}(n) = T_{r\_hyp}(n) = \frac{o^2}{o + (n-1)o} + (n-1)o = \frac{o}{n} + (n-1)o.$$

To further quantify the difference between sending/receiving communication times estimated by both models, we form  $T_{s\_hyp}/T_{s\_LogP}$  ( $T_{r\_hyp}/T_{r\_LogP}$ ):

$$\frac{T_{s\_hyp}}{T_{s\_LogP}} = 1 - \frac{n-1}{n^2},$$

which gives us the following bounds for  $n \geq 1$ :

$$\frac{3}{4} \leq \frac{T_{s\_hyp}(n)}{T_{s\_LogP}(n)} \leq 1. \quad (20)$$

## 6 Conclusions

A two-parameter hyperbolic model for parallel communication complexity on general dedicated networks has been proposed and validated by experiments with test programs containing communication patterns frequently encountered in scientific computations. The model captures both small-message and large-message timing behavior well, by design. Moreover, the quality of agreement between model and measurement at intermediate message sizes suggests that two parameters are adequate.

Each communication pattern, in principle, requires its own pair of parameters. The practical utility of the model in unstructured computations may therefore be limited. Fortunately, many scientific computations calling for parallel supercomputing rely on a small number of structured

communication patterns, so the work required to characterize the communication of an entire application remains tractable. Realistic analyses of communication can be used to influence algorithmic design, for a given architecture, and vice versa.

The hyperbolic model can be put into correspondence with the established  $(r_\infty, n_{1/2})$  model, but the new parameterization is more amenable to the construction of the reduction rules of Section 3. In the limit of small uniform messages that affords direct comparison with the LogP model, the hyperbolic and LogP models predict the same timings for elementary communication operations to within a factor of 3/4.

## 7 Acknowledgements

The cooperation of ICASE scientists in providing dedicated use of a subset of their Sparcstation Ethernet for the experiments is appreciated. The authors have benefited from many discussions with Professors Chet Grosch and Roger Hockney. Two of the authors of the LogP model have been helpful in clarifying aspects of our comparisons in Section 5. Anonymous reviewers also contributed greatly to this paper through detailed constructive comments.

## References

- [1] Alexandrov, A., Ionescu, M., Schauser, K. E., and Scheiman C. LogGP: Incorporating Long Messages into the LogP model - One step closer towards a realistic model for parallel computation, *7th Annual Symposium on Parallel Algorithms and Architectures*, Santa Barbara, CA, July 1995, to appear.
- [2] Butler, R., and Lusk, E. Monitors, Messages, and Clusters: The p4 Parallel Programming System, Argonne National Laboratory MCS Div. preprint P362-0493, and *Parallel Computing*, **20** (1994), 547–564.
- [3] Clark, D. D., Van Jacobson, Romkey, J., and Salwen, H. An Analysis of TCP Processing Overhead, *IEEE Communications*, June 1989, pp. 23–29.
- [4] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken T. LogP: Towards a Realistic Model of Parallel Computation, *SIGPLAN Notices*, **28** (1993), 1–12.
- [5] Cypher, R., and Leu, E. The Semantics of Blocking and Nonblocking Send and Receive Primitives, *Proc. of the International Parallel Processing Symposium '94*, Cancun, Mexico, IEEE Press, Los Alamitos, CA, 1994, pp. 729–735.
- [6] Dunigan, T. H. Performance of the Intel iPSC/860 and Ncube 6400 hypercubes. *Parallel Computing*, **17** (1991), 1285-1302.
- [7] Fortune, S., and Wyllie, J. Parallelism in Random Access Machines, *Proc. of the 10th Annual Symposium on Theory of Computing*, San Diego, CA, ACM Press, New York, 1978, pp. 114–118.
- [8] Hennessy, J. L., and Patterson, D. A. Computer Architecture - A Quantitative Approach, Morgan Kaufman, San Mateo, CA, 1990.
- [9] Hockney, R. W. Performance Parameters and Benchmarking of Supercomputers. *Parallel Computing*, **17** (1991), 1111-1130.
- [10] Hockney, R. W., and Curington, I. J.  $f_{1/2}$ : A Parameter to Characterize Memory and Communication Bottlenecks. *Parallel Computing*, **10** (1989), 277-286.
- [11] Hockney, R. W., and Jesshope, C. R. *Parallel Computers 2: Architecture, Programming and Algorithms*. IOP Publishing Ltd. (Adam Hilger), Bristol and Philadelphia, 1988.
- [12] Keyes, D. E. Domain Decomposition: A Bridge Between Nature and Parallel Computers, *Adaptive, Multilevel and Hierarchical Computational Strategies* (A. K. Noor, ed.), ASME, New York, 1992, pp. 293–334.
- [13] Leutenegger, S. T., and Sun, X.-H. Distributed Computing Feasibility in a Non-dedicated Homogeneous Distributed System, *Proc. of Supercomputing '93*, Portland, OR, IEEE Computer Society Press, 1993, pp. 143-152.
- [14] Maly, K., Khanna, S., Foudriat, E. C., Overstreet, C. M., Mukkamala, R., Zubair, M., Yerraballi R., and Sudheer, D. Parallel Communications: An Experimental Study, TR-93-20 Dept. of Comp. Sci., Old Dominion Univ., 1993.

- [15] Ramakrishnan, K. K. Performance studies in designing Network Interfaces: A Case Study, *Proc. of the 4th IFIP Conference on High Performance Networking '92*, A. Danthine and O. Spaniol, (Eds.). *Int. Fed. for Information Processing*, 1992, pp. F3-1 – F3-15.
- [16] Stallings, W. *Data and Computer Communications*, Macmillan, New York, 1991.
- [17] Stevens, W. R. *TCP/IP Illustrated vol. 1, The Protocols*, Addison-Wesley, Reading, MA, 1994.
- [18] Stoica, I., Sultan, F., Keyes, D. Modeling Communication in Cluster Computing. *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, February 15-17 1995, pp. 820-825.
- [19] Stoica, I., Sultan, F., Keyes, D. A Simple Hyperbolic Model for Communication in Parallel Processing Environments, *ICASE TR 94-78*, Institute for Computer Applications in Science and Engineering, September 1994.
- [20] Turcotte, L. H. A Survey of Software Environments for Exploiting Networked Computing Resources, Technical Report, Engineering Res. Ctr. for Computational Field Simulation, Mississippi State Univ., June 1993.
- [21] Valiant, L. G. A Bridging Model for Parallel Computation. *Communications of the ACM*, **33** (1990), 103-111.

## Biographies

Ion Stoica received M.S. degree in computer science from the Polytechnic Institute of Bucharest, Romania in 1989. He is currently a Research Assistant in the Computer Science Department at Old Dominion University. Before joining Old Dominion University in 1993, he worked as a Research Staff Member at the Research Institute for Informatics in Bucharest and as a Teaching Assistant at the Polytechnic Institute of Bucharest. His research interests include parallel and distributed algorithms, performance evaluation, and resource management in distributed systems.

Florin Sultan received a M.S. degree in Computer Science from the Polytechnic Institute of Bucharest, Romania, in 1989. He is currently a Research Assistant in the Computer Science Department at Old Dominion University. Before joining Old Dominion University in 1993, he worked as a Research Staff Memeber at the Research Institute for Informatics in Bucharest and as a Teaching Assistant at the Polytechnic Institute of Bucharest. His research interests include communication performance modeling and mobile computing.

David Keyes received a B.S.E. in Aerospace and Mechanical Sciences from Princeton in 1978, and an M.S. and Ph.D. in Applied Mathematics from Harvard in 1979 and 1984, resp. He is currently Associate Professor of Computer Science at Old Dominion University and Senior Research Associate at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center. He directs the Virginia/ICASE/LaRC Program in High Performance Computing and Communication, a graduate fellowship program hosted at ICASE. Keyes' research interests are parallel numerical algorithms for partial differential equations, parallel performance modeling, and scientific computing.