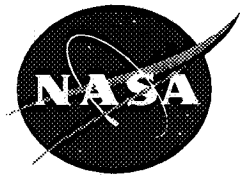
NASA Contractor Report 194984

ICASE Report No. 94-78



ICASE

A SIMPLE HYPERBOLIC MODEL FOR COMMUNICATION IN PARALLEL PROCESSING ENVIRONMENTS

Ion Stoica
Florin Sultan
David Keyes

Contract NAS1-19480
September 1994

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA 23681-0001

Operated by Universities Space Research Association

A Simple Hyperbolic Model for Communication in Parallel Processing Environments

Ion STOICA, Florin SULTAN, David KEYES*
Department of Computer Science
Old Dominion University
{*stoica, sultan, keyes*}@*cs.odu.edu*

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. Dedicated interprocessor links parameterized by a latency and a transfer rate that are independent of load are assumed by many existing communication models; such models are unrealistic for workstation networks. The communication system is modeled as a directed communication graph in which terminal nodes represent the application processes that initiate the sending and receiving of the information and in which internal nodes, called communication blocks (*CB*s), reflect the layered structure of the underlying communication architecture. The direction of graph edges specifies the flow of the information carried through messages. Each *CB* is characterized by a two-parameter hyperbolic function of the message size that represents the service time needed for processing the message. The parameters are evaluated in the limits of very large and very small messages. Rules are given for reducing a communication graph consisting of many *CB*s to an equivalent two-parameter form, while maintaining an approximation for the service time that is exact in both large and small limits. The model is validated on a dedicated Ethernet network of workstations by experiments with communication subprograms arising in scientific applications, for which a tight fit of the model predictions with actual measurements of the communication and synchronization time between end processes is demonstrated. The model is then used to evaluate the performance of two simple parallel scientific applications from partial differentialequations: domain decomposition and time-parallel multigrid. In an appropriate limit, we also show the compatibility of the hyperbolic model with the recently proposed LogP model.

*The third author's research was supported by the National Aeronautical and Space Administration under NASA contract No. NAS1-19480 while in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681-0001.

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 parallel/distributed 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.

The issue of communication is only recently beginning to receive attention in keeping with its importance in models of parallel computation. Most parallel models following the precedent of [6] start with the assumption of "perfect" communication, namely no delay and unlimited bandwidth. Algorithms based on such models may appear to be highly performant, but more realistic assumptions [4] about the underlying communication system reveal significant degradation of their behavior.

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_1 Messages exchanged between end processes are not corrupted.

A_2 No duplicates of transmitted messages are generated.

A_3 Between any pair of end processes, messages are received in the order they were sent.

A_4 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 often not taken into account. Instant communication (implying a communication delay equal to zero) is assumed. A common idealization is to assume that an unlimited number of processors can use unlimited bandwidth.

Besides these considerations about the theoretical approaches to parallel computing, our approach is 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. It is unrealistic to assume that communication is bounded as more data are stored and processed on each node. On the other hand, 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 the main 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 publically available systems for workstation collaboration are annotated in [18]. 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. However, in the case of the tightly coupled parallel computers, where these properties can be supported directly by the hardware (through a highly reliable interconnection network and simple hardware protocols), we assume that no other requirement is enforced on the communication links. That is, we assume that only the minimal requirements of a reliable communication are met and no specific protocols supporting other costly facilities are implemented in addition.

We focus throughout this paper on the general case of communication within a cluster of workstations cooperating in a distributed computation. Tightly coupled multiprocessing is included, and we analyze how it compares with LogP model [4] in the last section. However, 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 non-dedicated 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.

Another important problem when evaluating the performance of parallel algorithms in practice is the distribution of work among processing nodes. This is important in multitasking environments (especially in distributed computing) where the processing nodes are not guaranteed to be available at all times. This impacts not only 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 fluctuating loads 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. An interesting report exploring the opposite extreme (interfering end processes, but free communication) is [13].

This paper is organized as follows:

Section 2 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 3 describes communication patterns generated by operations common in parallel algorithms (broadcast, global operations, synchronization and nearest neighbor communication) and describes how these can be evaluated using the model. Results obtained in experiments with these operations in a distributed computing environment are presented as a first step in validating the model. Predictions and experiments disagree by at most 15% over a range of message sizes from one byte to 64Kbytes, on up to 16 dedicated workstations connected by Ethernet.

Section 4 presents a method for determining the parameters of the communication blocks when modeling communication in a cluster of workstations.

Section 5 describes two model parallel scientific applications used to test our model, giving rise to various communication patterns as well as a wide range of message size distributions and communication to computation ratios. Results of the experiments support the model in fitting with predictions of the cost of communication between end processes, to within the limits of control of interprocessor synchronization.

Section 6 describes the LogP model of computation for massively parallel processors [4] and shows favorable comparison of the hyperbolic model with it in the small message regime, for which the comparison is easily made.

2 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;*
- *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 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 *CB*s are **dependent** iff only one of them can process data at a moment in time and **independent** iff at any moment in time all *CB*s can process *different* streams of data without interfering. For example, two *CB*s 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, transmission of the packet on the communication network.

As an 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 corresponding *CG* of 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 send one byte of data (called *transmission time*). The incremental service time includes the overhead generated by the protocol layers to ensure the reliability (e.g., acknowledgment packets).

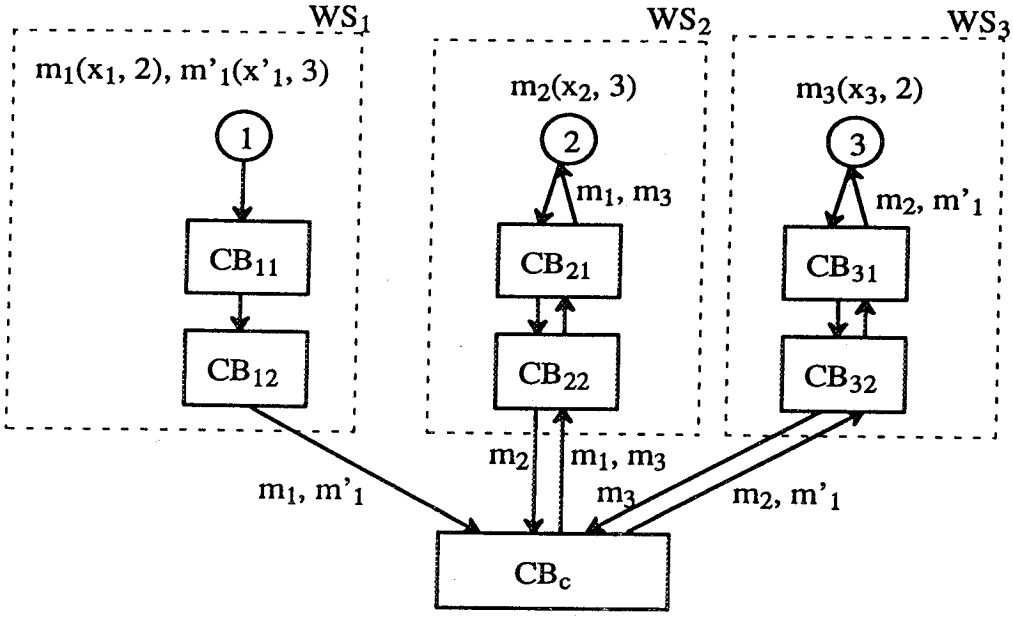


Figure 1: *Communication graph (CG) for three processes running on different workstations. Each terminal node (process) contains the list of messages it sends (node 1 sends m_1 to 2 and m_1' to 3, node 2 sends m_2 to 3 and node 3 sends m_3 to 2). Each edge is labeled with the list of messages that flow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

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

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

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

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

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 $x \to 0$, $t(x; a, m, p) \to a$ and for $x \to \infty$ (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

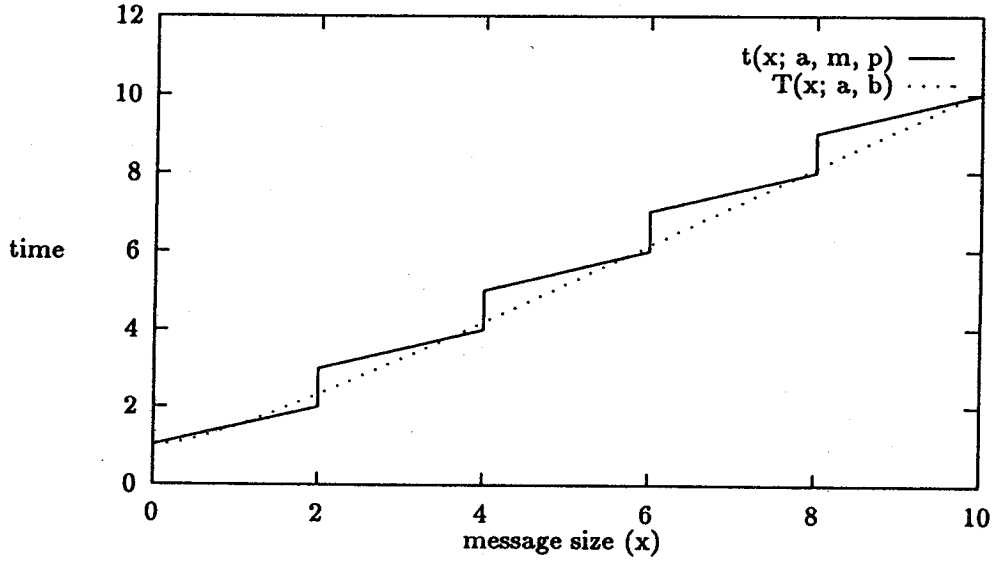


Figure 2: *The total service time t(x;a, m, p) versus the continuous function T(x; a, b) used to approximate it (a = 1, m = 0.5 and p = 2).*

time t with the following monotonically increasing continuous function defined on the interval $[0, \infty)$ (Figure 2):

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

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.

The improvement of (3) over a simple latency (α) / reciprocal transfer rate (β) model,

$$T(x; \alpha, \beta) = \alpha + \beta x, \tag{4}$$

is not so much in the fit of a continuous curve to the sawtooth form of a packetized transmission, but in the analytical simplicity with which the parameters (a, b) for a CG may be derived in terms of its elemental CBs, as shown by the four combination rules in subsections 2.1 through 2.4. Using T_i to estimate the total service time required by CB_i (now 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 CBs interconnected in various structures to a single equivalent CB, with service time $T(a_1, b_1, a_2, b_2, \ldots, a_n, b_n)$.

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

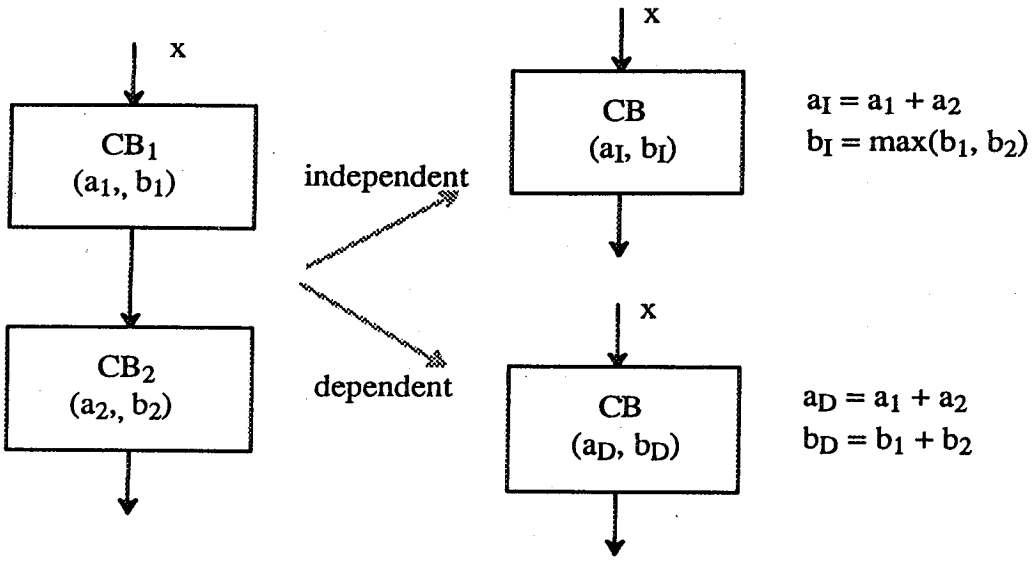


Figure 3: *The equivalence transformation for two serial interconnected independent CBs (right top) and dependent CBs (right bottom).*

subsequent packets of the same message. The corresponding *CB* has the following parameters:

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

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 $a^{(1)} - a^{(2)}$.

2.1 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.*

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 *CBs* 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* which 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 CBs . 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 $x \to \infty$ the dominant term in the total service time is $\max(b_1 x, b_2 x)$, 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 $x \to 0$, 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 :

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

2. Dependent CBs . Here, 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 $b_1 x + b_2 x$. This gives us the following CB parameters:

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

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:*

$$a = \sum_{i=1}^{n} a_i; \;\; b = \max(b_1, b_2, \ldots, b_n) \tag{5}$$

if all CBs are independent, and

$$a = \sum_{i=1}^{n} a_i; \;\; b = \sum_{i=1}^{n} b_i \tag{6}$$

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 serial 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 serial 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: $a = a_a + a_{os} + a_c$, $m = \max(b_a + b_{os}, b_c)$.

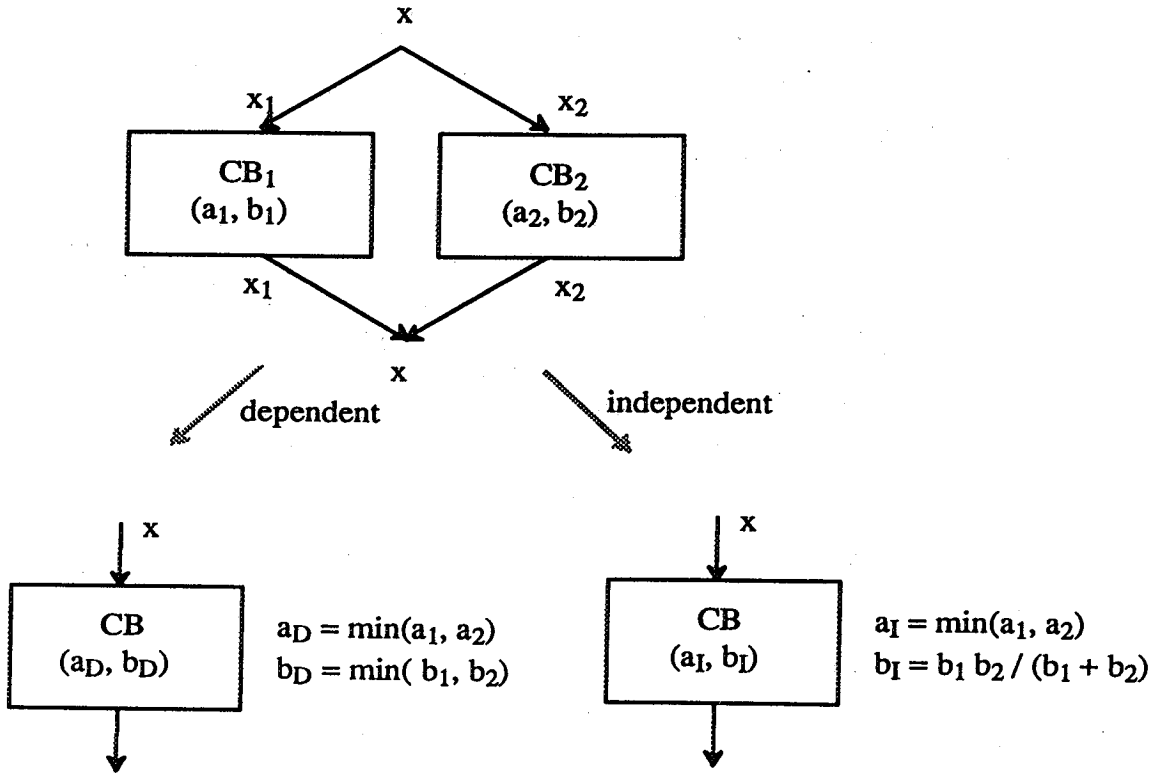


Figure 4: *The equivalence transformation for two parallel interconnected dependent CBs (bottom left) and independent CBs (bottom right).*

2.2 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 $x \to 0$ 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 $x \to \infty$, 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 sizes of the inputs processed by CB_1 and by CB_2 respectively, it is easy to see that load balancing is achieved when $x_1 = xb_2/(b_1 + b_2)$, $x_2 = xb_1/(b_1 + b_2)$. Finally, combining either of these solutions with the asymptotic expression of the total service time of CB, $T(x; a, b) = bx$ for $x \to \infty$, we obtain the overall CB parameter set:

$$a_I = \min(a_1, a_2); \;\; b_I = \frac{b_1 b_2}{b_1 + b_2} \tag{7}$$

2. Dependent CBs . Since both CBs 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 $x \to 0$

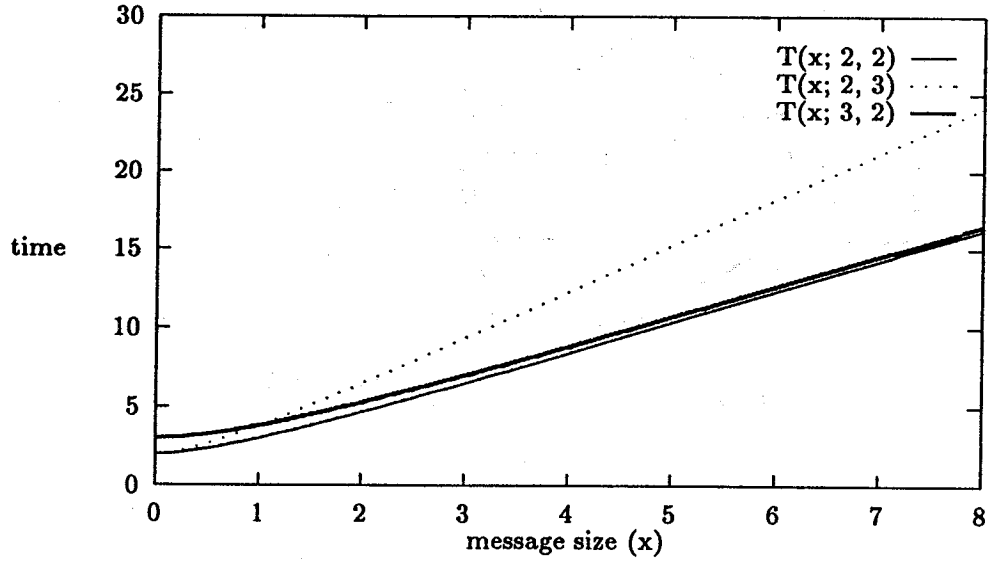


Figure 5: *The total service time T(x; 2, 2) for the resulting CB after the equivalence transformation of two dependent parallel interconnected CBs with total service time given by T(x; 2, 3) and T(x; 3, 2), respectively.*

we choose the *CB* that has the minimum fixed service time, while for $x \to \infty$ we choose the *CB* that has the minimum incremental service time), which gives us the following results:

$$a_D = \min(a_1, a_2); \;\; b_D = \min(b_1, b_2) \tag{8}$$

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:*

$$a = \min(a_1, a_2, \ldots, a_n); \;\; \frac{1}{b} = \sum_{i=1}^{n} \frac{1}{b_i} \tag{9}$$

if all CBs are independent, and

$$a = \min(a_1, a_2, \ldots, a_n); \;\; b = \min(b_1, b_2, \ldots, b_n); \tag{10}$$

if all CBs are dependent.

Figure 5 shows the total service time functions of two dependent parallel interconnected *CBs* and of the resulting *CB*, after the equivalence transformation. The initial *CBs* have the communication parameters $a = 2$, $b = 3$ and $a = 3$, $b = 2$ respectively. According to the above rule, in this case (dependent interconnection), the equivalent *CB* has as parameters $a = \min(2, 3)$, $b = \min(3, 2)$. At the limits, the total service time function of the equivalent *CB*, $T(x; 2, 2)$, approaches asymptotically the better of the service time functions of the initial *CBs*, i.e. for $x \to 0$ approaches $T(x; 2, 3)$, while for $x \to \infty$ approaches $T(x; 3, 2)$.

Before turning to the more difficult case of concurrent message processing, we summarize the results of serial and parallel interconnection on independent and dependent *CB*s. In the small message limit that governs the a parameter, *CB*s in serial combine additively and *CB*s in parallel combine by taking the minimum. In the large message limit that governs the b parameter, *CB*s in serial that are dependent combine like resistors in series, and *CB*s 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.

2.3 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, ..., m_n of sizes x_1, x_2, ..., x_n to be processed (Figure 6). We assume that *CB* processes m_1, ..., 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, ..., i_n is a permutation of 1, ..., 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 $x_i \to 0$, $(i = 1, \ldots, n)$. 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 $x_i \to \infty$. 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:

$$t(x_1, x_2, \ldots, x_n; a, b) = b \cdot \sum_{i=1}^{n} x_i. \tag{11}$$

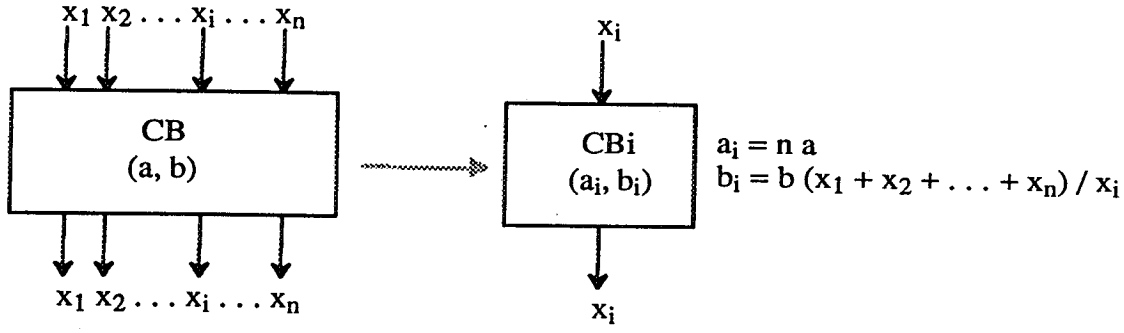


Figure 6: *The equivalence transformation in the concurrent message processing case.*

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:

$$T(x_i|X, n; a, b) = \frac{(na)^2}{na + bX} + bX, \tag{12}$$

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 (12) as:

$$T(x_i; a, b) = \frac{a'^2}{a' + b'x_i} + b'x_i \tag{13}$$

By associating (13) 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, ..., m_n of sizes x_1, x_2, ..., x_n, respectively, is equivalent to a structure of n communication blocks $CB_1(a_1, b_1)$, $CB_2(a_2, b_2)$, ..., $CB_n(a_n, b_n)$, where CB_i independently processes the message m_i and has parameters:*

$$a_i = na; \;\; b_i = b \cdot \frac{\sum_{i=1}^{n} x_i}{x_i} \tag{14}$$

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 7, 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 7(a)) in two steps. First, applying rule 3 to CB_c and CB_4 we obtain an intermediate structure consisting of three serial interconnected independent communication blocks $CB_1(a_1, b_1)$, $CB'_c(a_c, b_c)$ and $CB'_4(a'_4, b'_4)$ (Figure 7(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 7(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 equation (3).

2.4 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. In consequence 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:*

$$T(x; a, b) = \frac{a^2}{a + bx} + bx \tag{15}$$

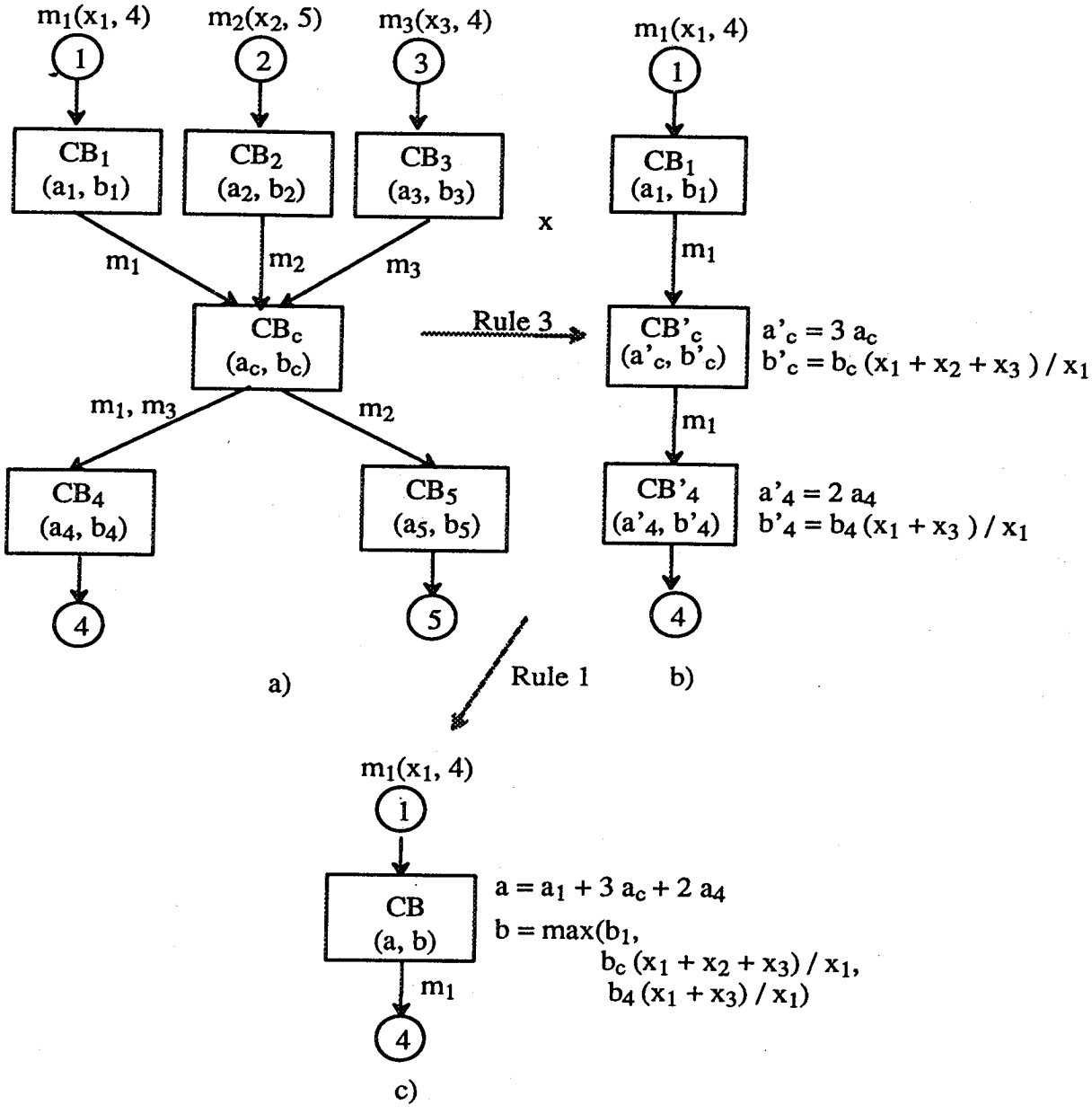


Figure 7: *Computing the estimated time for the message m_1 to be delivered to the process 4 by successive reductions of the communication graph.*

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 CGs. 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.

2.5 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 CBs : CB_1, CB_2, ..., CB_n (we consider that this is always possible), where the source process runs on the same processor as CB_1 and the destination process runs on the same processor as CB_n. 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 CBs . If the message is large, i.e. it consists of a large number of packets, then the message is concurrently processed by the independent CBs 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.

In our model we define T_s either as the time required to process all message packets by CB_1, or as a 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 redefining overloaded terms, such as *blocking/non-blocking* and *synchronous/asynchronous*, which are usually 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 calls overhead, then it is clear that 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 Figure 8):

- 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 it is easy to see that 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 to 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 as parameters:

T_s :

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

T_r :

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

Figure 8: *Sending (T_s) and receiving time (T_r) expressions, where message size is x.*

a, equal to the average time interval to receive the acknowledge from receiver by the sender, 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_c, T_r is defined either as the time required to process all packets of a message by CB_n (*preemptive* **receive** primitive), or as a 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). The T_r analysis is the same as T_s analysis for small messages and for large messages when *preemptive* **receive** primitive is used (see Figure 8). The major difference is when we consider large messages and *preemptive* **receive** primitives. Unlike the *preemptive* **send** primitive, where after the first packet is sent the application process can proceed, when receiving we assume that the application cannot proceed until the last packet of the message is received (in other words, the message is not passed to the application process until it is completely received) and thus we take T_r equal to T_c. 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 in Figure 8 only the expressions of T_s and T_r for the extreme message sizes ($x \to 0$, $x \to \infty$) are given, we can use again the equation (3) to approximate $T_s(x; a_s, b_s)$ and $T_r(x; a_r, b_r)$ for any message size x, where $a_s = T_s(x \to 0)$, $a_r = T_r(x \to 0)$ and $b_s = \lim_{x\to\infty} \frac{T_s(x)}{x}$, $b_r = \lim_{x\to\infty} \frac{T_r(x)}{x}$.

3 Common Communication Patterns in Parallel Applications

In this section we give some examples of 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 9 shows the communication graph for a message transmission between two processors.

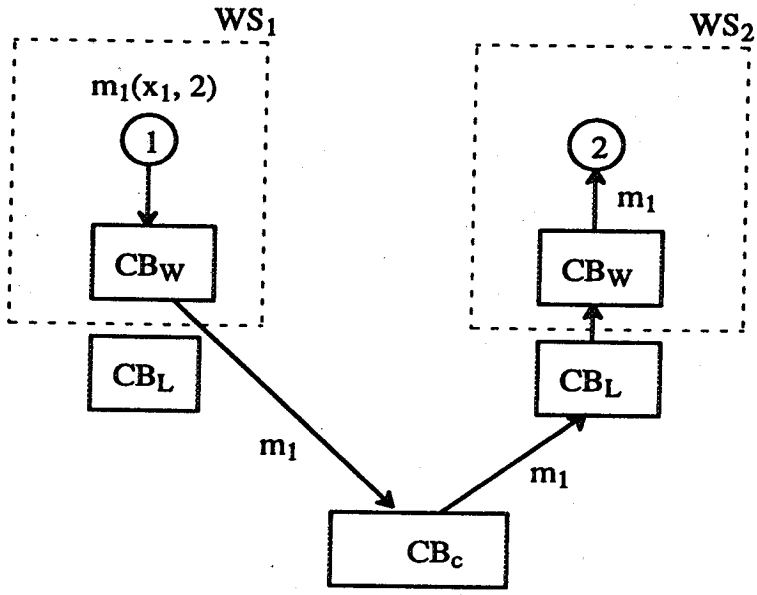


Figure 9: *The communication graph for sending one message from process* 1 *to process* 2. *Observe that the* CB_L *appears in the communication path only between communication network* (CB_C) *and workstation communication block* CB_W.

3.1 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 10. 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 ($x \to 0$), and the message is very large ($x \to \infty$). 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 it is easy to see that the communication time between any two processors is $T_c = 2a_W + a_C + a_L$, while the sending and receiving times are $T_s = T_r = a_W$. Back to our example (Figure 10(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 10(c). Since we consider *non-preemptive* **send** and **receive** primitives, we have $T_s = T_r = T_c$ (see tables in Figure 8). As one can observe 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 (from our topology a message travels along only one path between source and destination) 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 broadcast in our example is equal to $x(2\max(b_C, b_W) + \max(2b_C, b_W) + 2\max(3b_C, b_W))$. Now, the entire broadcast communication graph can be reduced to one communication block CB_{BCAST} with the following parameters: $a_{BCAST} = 8a_W + 3a_C + 3a_L$, $b_{BCAST} = 2\max(b_C, b_W) + \max(2b_C, b_W) + 2\max(3b_C, b_W)$.

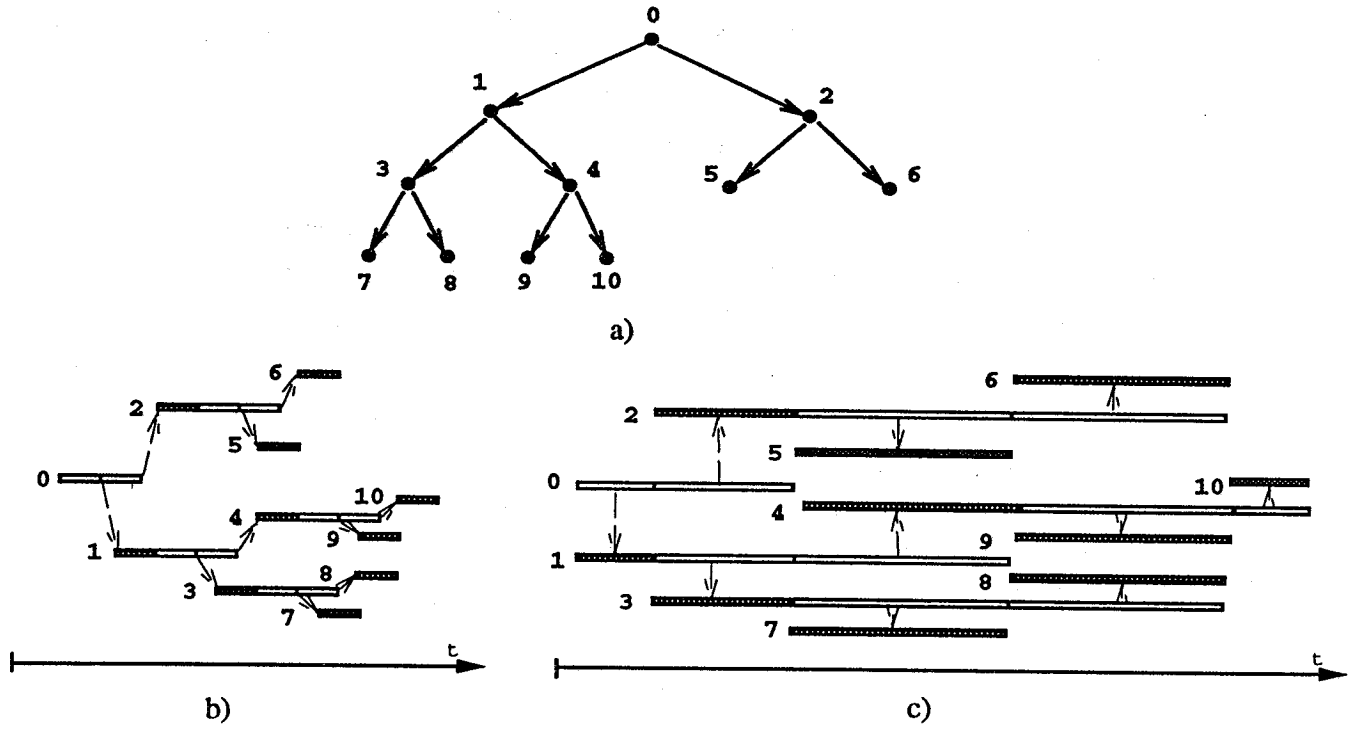


Figure 10: *a) The broadcast binary tree for 11 processors. b) The processor activity when a small message is broadcast. The sending time is represented by empty bars while the receiving time is represented by shadowed bars. c) The processor activity when large messages are broadcast.*

In the second broadcast implementation (which is the native implementation in p4, version 1.3) the root processor simply sends the message to every other processor: 1, 2, ..., N. It is very easy to see that in this case the time to complete the broadcast for small messages is $(N+1)a_W + a_C + a_L$ and for large messages is $N \cdot \max(b_C, b_W) \cdot x$. 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 (this can be easily verified for the binary-tree broadcast implementation). This is because 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

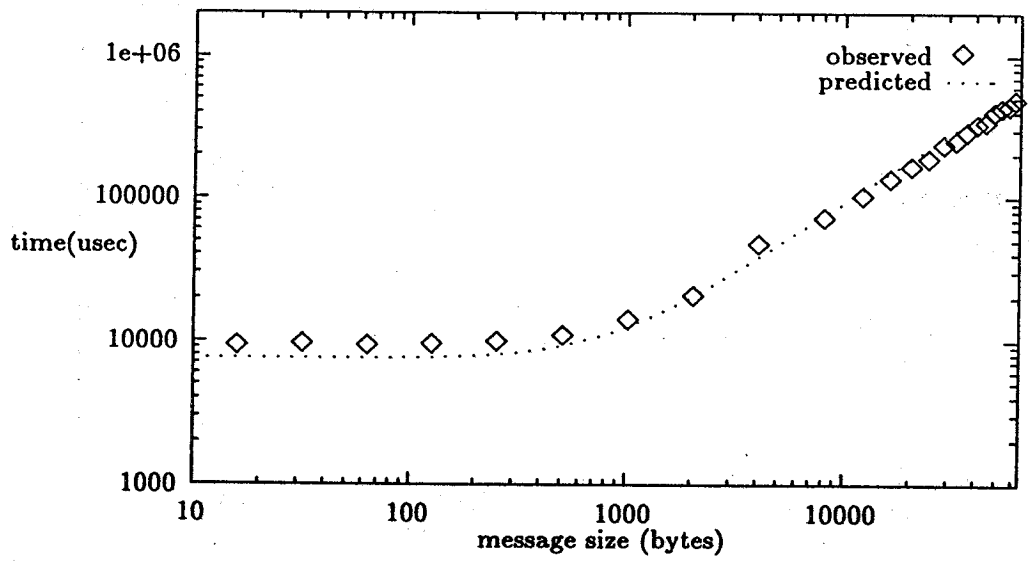


Figure 11: *The estimated T_c versus experimental data for the broadcast binary tree implementation. The regression coefficient is 0.88. The experiments were run on 11 Sun SparcstationELC workstations interconnected by an Ethernet network using p4.*

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.

The shapes of the estimated communication time functions for the tree-based and serial broadcasts, together with experimental measurements of T_c are shown in Figures 11 and 12, respectively. All experiments in this and subsequent sections 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. The model offers a very accurate approximation to the actual measurements. The regression coefficients are 0.88 for the binary tree broadcast implementation and 0.92 for the second broadcast implementation. In both cases, the maximum error was around 10%. All *CB* parameters were experimentally determined in the limits of small or large message size and one or many processors, using the procedure described in section 4.

3.2 Synchronization and Global Operation

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, the synchronization can be seen as a special case of global operation where no computation is performed. In the remaining of this subsection, we concentrate on the global operation implementation.

A global operation primitive implements a group computation. Formally, a group compu-

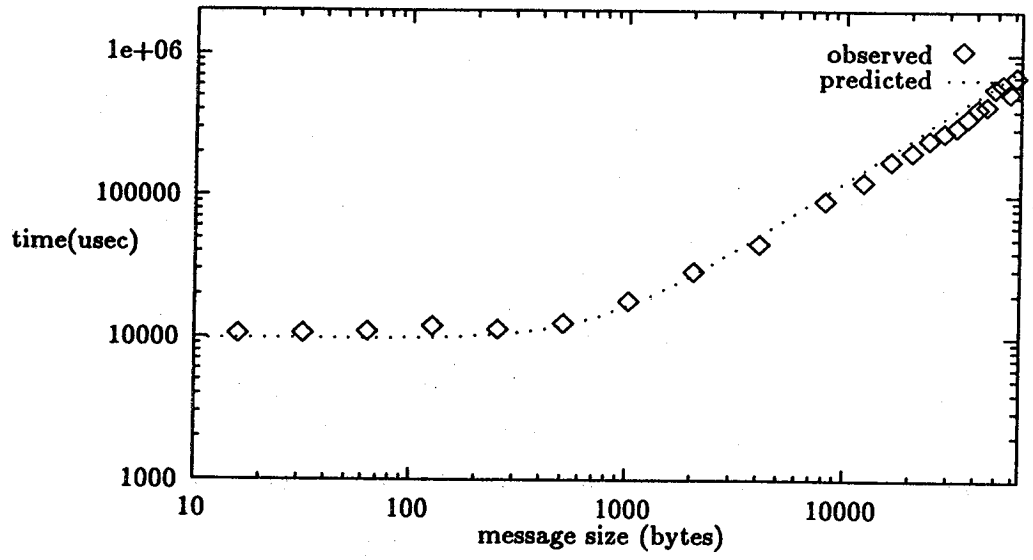


Figure 12: *The estimated T_c versus experimental data for p4 native broadcast implementation. The regression coefficient is 0.92.*

tation is defined as follows: given n different items a_1, a_2, ..., a_n in a group $(S, \oplus)$ (where $\oplus$ is a binary associative and commutative operation defined on set S) compute the final value $a_1 \oplus a_2 \oplus \ldots \oplus a_n$. The following are examples of group computation: finding the sum, the maximum, or the minimum of a set of n numbers.

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 13(a), which is from the same incomplete binary tree used in the broadcast illustration. 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.

Since the values carried by the messages are often no larger than 8 bytes (double precision numbers), 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 communication time. From Figure 13(b) it may be seen that the total time required to complete the global operation is $19a_W + 4a_C + 4a_L$. The observed 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.)

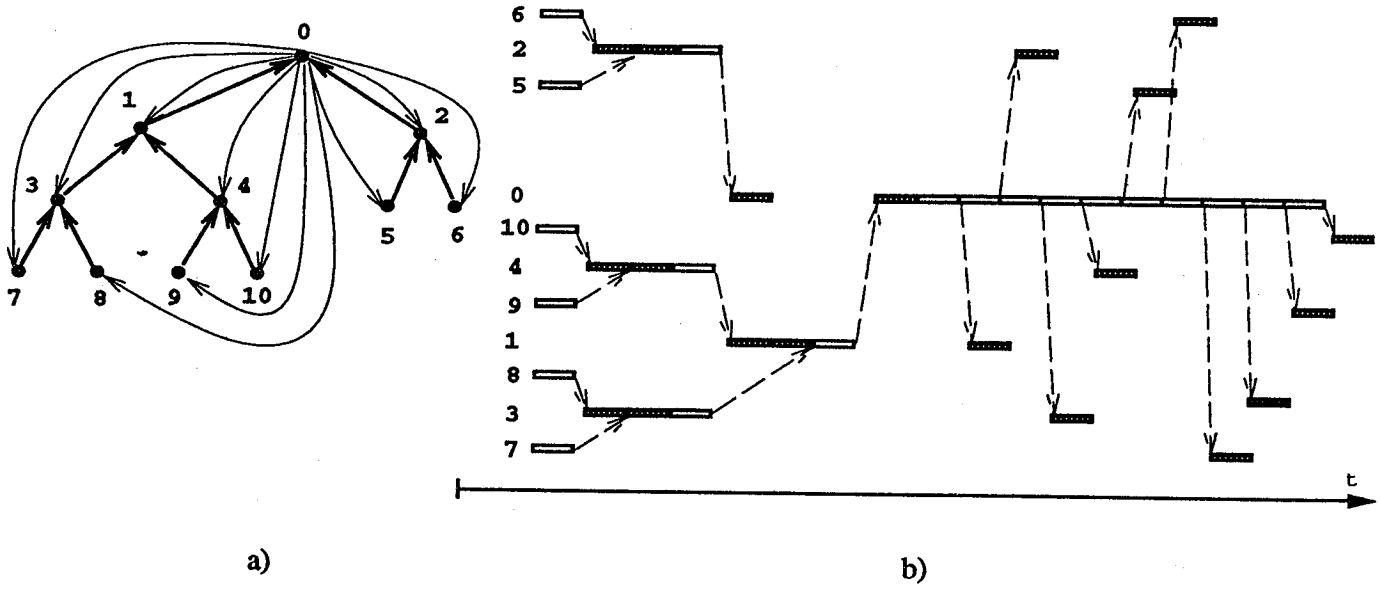


Figure 13: *a) The communication pattern used by the syncronization and global operation primitives for 11 processors. b) The processor activity over time.*

3.3 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 ([11]; see section 5.1) 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 is based on a parallel application model in which every processor has the same amount of work to perform between 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 14 shows the estimated T_c function versus experimental measurements with a regression coefficient of 0.92, and a maximum error of 17%.

4 Experimental Evaluation of CB parameters

In principle, one can determine CB parameters by considering a workstation's physical characteristics (e.g., the processor speed, the memory access time, the internal bus speed, etc.), and the communication protocol implementation details (e.g., how many times a data buffer is copied while passed through various protocol layers, the algorithms used to compute the checksum, etc.). Although this approach apparently allows accurate evaluation of CB parameters, it is very hard to apply in practice because of several factors:

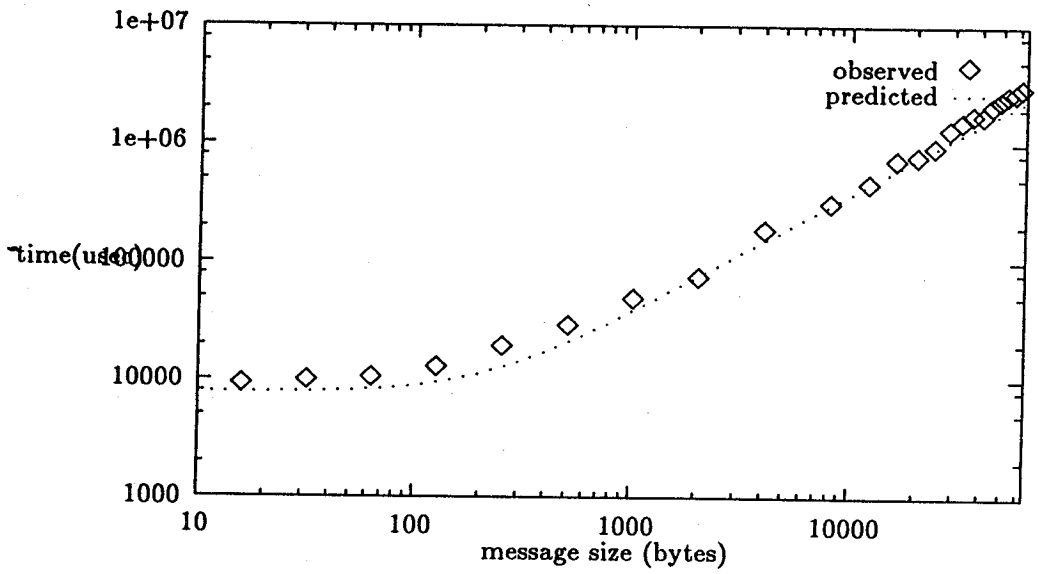


Figure 14: *The estimated T_c versus experimental data for neighbor communication pattern. The regression coefficient is 0.92. Here, $N = 16$ and $K = 4$.*

- Various layers of the communication architecture are embedded in the general purpose operating systems running on the processing nodes. This makes them compete for system resources with other processes in the multitasking environment. It also means that various factors like interrupt processing, context switching, memory management, etc., combined with hardware features like the presence of a cache memory system, would have to be considered when trying to model the communication.
- Systems may be heterogeneous (made up of machines from different vendors, with different characteristics and running different operating systems).
- Software packages, such as the support for communication between end processes (at the application level), each having their own characteristics and introducing their own overhead, which would have to be represented in a detailed model.

We propose here a simple approach to evaluate the CB parameters with an accuracy whose acceptability can be judged by its fits in Figures 11, 12 and 14. Let us consider a network of n identical workstations linked by a communication network $CB_c(a_c, b_c)$. For simplicity, assume that the overhead for sending and receiving messages is equal. Thus, all workstations are modeled by the same $CB(a, b)$ irrespective of whether a message is being sent or received. Now, consider $2n$ workstations numbered from 1 to $2n$, and let each odd numbered workstation send a message of the same length to the next even numbered workstation, i.e. $2i - 1$ sends to $2i$, $(i = 1, 2, \ldots n)$. If we take a pair of workstations $2i - 1$ and $2i$ and first apply rule 3 for CB_c and next rule 1 for $CB_{(2i-1)}$, $CB_{(2i)}$ and CB'_c, the total service time required to deliver the message from $2i - 1$ to $2i$ is given by:

$$T(x, n; a', b') = \frac{a'^2}{a' + b'x} + b'x \tag{16}$$

where $a' = 2a + na_c$ and $b' = \max(b, b_c n)$.

There are four parameters to be determined: a and b for the workstation CB and a_c and b_c for the network CB_c. Theoretically, we can determine all necessary parameters from the following equations:

$$\begin{aligned} a_c &= \lim_{n\to\infty} \frac{a'}{n} = \lim_{n\to\infty} \frac{\lim_{x\to 0} T(x, n; a', b')}{n} \\ b_c &= \lim_{n\to\infty} \frac{b'}{n} = \lim_{n\to\infty} \frac{\lim_{x\to\infty} \frac{T(x,n;a',b')}{x}}{n} \\ a &= \frac{a' - na_c}{2}\,|_{n=1} = \frac{\lim_{x\to 0} T(x, 1; a', b') - a_c}{2} \\ \max(b, b_c) &= \lim_{x\to\infty} \frac{T(x, 1; a', b')}{x} \end{aligned} \tag{17}$$

Notice that the last equation permits determination of b, the reciprocal of the bandwidth of the CB associated with each workstation, only if it is larger than b_c. If it is smaller than b_c, it is unnecessary, since the workstation CB is then not the bottleneck in the large message limit. The first two equations express the well-known truth that when the number of workstations increases, the network becomes the main bottleneck for the overall performance.

For the SparcstationELCs running SunOS 4.1.3, the p4 communication layer version 1.3, and the Ethernet at ICASE, where the experiments were performed during "dedicated" wee hours, the parameters we obtained and used in the "predicted" curves in this paper are:

$$\begin{aligned} a_c &= 345.60\ \mu\text{sec} \\ b_c &= 0.92\ \mu\text{sec/byte} \\ a &= 859.52\ \mu\text{sec} \\ b &= 1.42\ \mu\text{sec/byte} \end{aligned}$$

We note that $1/b_c$ is only about 10% slower than the theoretical peak performance of Ethernet, virtually the same performance realization reported in [17]. We expect the b parameter of the workstation to be visible only when there is low contention, since it is within a factor of two of the reciprocal of b_c.

5 Tests on Model Scientific Applications

Two model parallel scientific applications originally written for a tightly coupled multiprocessor and rewritten in p4 are used as test programs for the hyperbolic model. A domain decomposition (DD) code for the Poisson problem on the unit square and a multigrid (MG) code for transient flow in a cavity are chosen among conveniently available codes for their simplicity and for their very different communication patterns. For each application, we first describe the algorithm just sufficiently to expose the leading order computational and communication complexity and to appreciate its general context, then we describe the network parallel implementation. Fuller descriptions of the applications themselves may be found in references [12] and [9]. For each application, we select for graphical comparison various communication cost estimates and corresponding measurements. The estimates derive from appropriate combinations of the archetypal communication operations described in section 3, with parameters evaluated as in section 4.

5.1 A Domain Decomposition Application

5.1.1 Algorithm

The first test problem is a partial differential equation (Poisson's equation) on a two-dimensional square domain with given boundary conditions and a forcing term chosen so that the solution is smooth. We assume uniform gridding and discretize with the standard five-point difference stencil. This generates a sparse banded system of linear equations. A domain decomposition method using conjugate gradient (CG) iteration is used to solve the resulting matrix equation. The domain is divided into uniform square subdomains by the vertices of a coarse grid (nested in the grid on which the problem is resolved), and by the edges connecting these vertices. Altogether, three point sets are distinguished: the coarse grid vertices (or crosspoints), the fine grid points along the edges (or interfaces), and the fine grid interior points. The union of the vertices and edges is called the "wire basket." The decomposition of the physical domain induces a block structure on the system matrix. The CG iteration is over the full set of unknowns.

The code used in the tests was originally written for the Intel hypercube by Keyes & Gropp [12] and uses a BPS-type wire-basket preconditioner [1]. The preconditioning consists of several serial stages, most of which permit concurrency with a granularity equal to the number of subdomains. Independent problems on the subdomain interiors are solved concurrently in each CG iteration. The only communication needed to set them up is in supplying values along the four bounding segments, which may be segments of the physical boundary or artificial interior interfaces. Independent problems on the interfaces are solved once per CG iteration. The only communication required to set them up is in supplying forcing data along the interfaces from adjacent subdomain interiors. The remaining stage is the solution of a small but global linear system involving the coarse grid unknowns; this is the only exception (besides load imbalance due to boundary effects) to full concurrency in the application of the preconditioner. Rather than solve this problem in a true distributed fashion, which is cost-effective only for small numbers of processors, the coarse grid problem is assembled *in full*, redundantly, on each processor, and then solved serially. Global all-to-all communication is required to set up the corresponding right-hand side, whose values change at each iteration, and whose computation, in turn, requires values from along the four interfaces that meet at each crosspoint. The interior and interface phases involve local communication whose overall volume scales with both problem size and granularity. The coarse grid phase involves low-bandwidth global broadcasts whose number (one for each crosspoint) scales with the granularity, but not with the problem size. A detailed analysis of parallel implementations of methods of this type on tightly-coupled distributed memory machines, such as hypercubes, meshes, or rings, is given in [7]. Such machines have multiple direct links, whose number scales with the number of processors, so that the local interior and interface communications scale perfectly.

5.1.2 Implementation

On an Ethernetwork of workstations, all communication phases compete for a common resource. The degree to which they collide depends on the volume and on the synchronicity of the messages. Figure 15 shows the most general communication pattern generated for an internal subdomain with n_i and n_j points per vertical and horizontal side, respectively. Obviously, a processor that holds a subdomain located on the physical boundary need not participate in

the complete set of messages shown. The numbers assigned to incoming or outgoing messages define the order of communication operations, as imposed by the data dependencies in the algorithm. Each message is labeled with its type and the number of data elements (floating point values) carried.

One all-to-all set of broadcasts is performed per iteration to distribute the crosspoint system to all of the p processors. Two additional global reduction operations (not represented in the figure) are executed at each iteration as part of the CG algorithm itself, independent of the the preconditioning. These operations are the inner products that compute the step lengths in the vector updates of CG algorithm. The inner product arithmetic scales as the problem size, but the message volume for these global reductions scales with the granularity only, since the contribution from each processor is condensed to a scalar with local operations before any data is shared. The global broadcasts and reductions have a "self-synchronizing" effect on the parallel algorithm. The main outcome of frequent synchronization is that most of the measured communication time is spent by processors that finish their computations early idling while waiting to receive messages from those that are delayed.

Three main characteristics of the communication requirements of the algorithm are: the communication pattern employed is independent of the data, the number of messages exchanged and the number of global operations per iteration are independent of the iteration number, and the size of the messages exchanged between neighboring processors is independent of the number of processors. These characteristics permit very simple analytical models of parallel complexity and scalability [7], since most aspects of the computation and communication can be estimated by considering a single processor on a single iteration.

To estimate the computation and communication complexity, consider that the subdomains are logically square, i.e. $n_i = n_j = n$. Since the algorithm requires that only the points on the boundary be exchanged between adjacent subdomains, the communication complexity is $\mathcal{O}(n)$ per subdomain. On the other hand, the computation complexity of the algorithm for each subdomain is $\mathcal{O}(n^2)$ for the unpreconditioned CG method (a fixed number of stencil operations at each point) and $\mathcal{O}(n^2 \log n)$ for an FFT-based fast elliptic solver used on each subdomain. When the computational work increases we expect that any irregularities between the timings of identical phases of the computation performed on different processors will also increase with the same power law, i.e. $\mathcal{O}(n^2)$ or greater. Thus, as n increases the differences between the moments when messages sent by different processors physically hit the network increase much faster ($\mathcal{O}(n^2)$) than the message transit times on the network ($\mathcal{O}(n)$). Therefore, we expect that for large n there will be practically no contention for the physical network. The dominant cause for degradation of efficiency in the large n limit is synchronization. In the opposite small n limit the messages that are sent are tiny and the actual transmission time on the network is much smaller than the sending/receiving overheads. Therefore, the dominant cause for degradation of efficiency in the small n limit is latency.

The BPS DD algorithm is run on up to 16 SparcstationELC workstations for the following subdomain sizes: 16, 32, 64, 128, 256, 512. The largest of these problems corresponds to a square containing $(4 \times 512)^2 \approx 4.19 \times 10^6$ grid cells and thus to a matrix containing approximately 2×10^7 nonzero real entries. As partial differential equation discretizations go, this is a large problem. Figures 16 and 17 compare predicted and experimental timings for all 16 workstations, in two types of tests. In the first test, the original code is run without any synchronization beyond that inherited from the algorithm itself. We examine here the sending time for the total of all messages per iteration, averaged over all processors and all iterations,

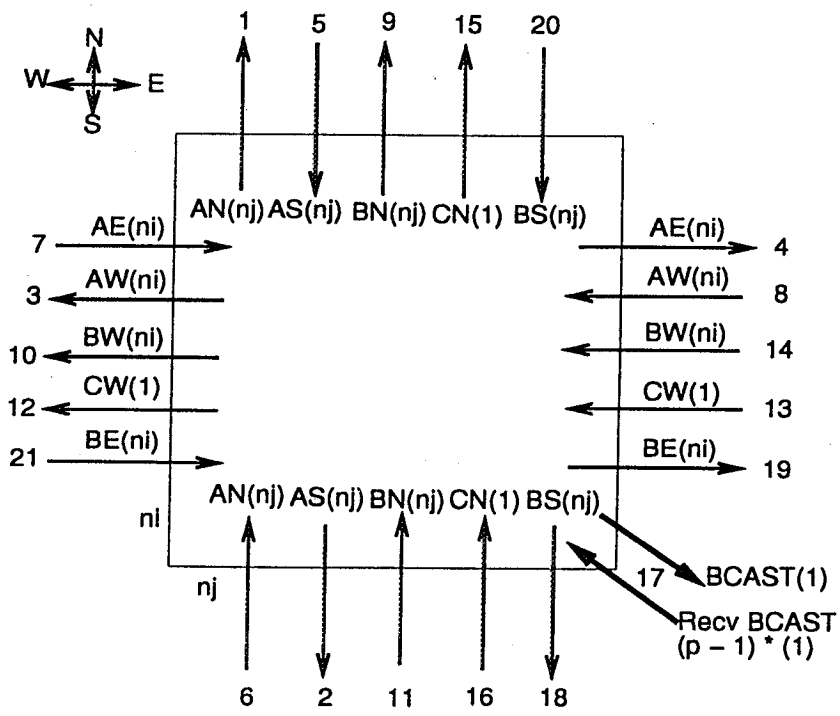


Figure 15: *Communication pattern for a processor associated with an internal subdomain in the DD method applied to the Poisson problem. A, B, and C are distinct message types, the compass points indicate message directions, and the number in parentheses is the message size in real words.*

as a function of subdomain size. In the absence of a global clock, it is impossible to measure the actual message transit time, which would be a difference of absolute times on two different processors. Instead, we use the sending time (T_s in the notation of section 2.5). It was shown in section 2.5 that for the *non-preemptive* **send** primitive, the sending time is equal to the total communication time minus the transmission and processing time of the last packet of the message at the receiver. The sending (resp., receiving) time is defined as the difference of absolute times measured on the same clock immediately before and after posting a send (resp., receive), blocking, and returning. Figure 16 shows the predicted and measured sending times, with a maximum error of about 20%. Since many of the messages are small (one real word), overall dependence on subdomain size, which shows up only in the vector messages transmitting boundary data, is weak.

Next, we modify the application by inserting additional synchronization points so that all neighbor communications wait until all required data is computed and ready, and we measure the time interval between the synchronization moment and the moment when the nearest-neighbor vector messages (only) are received by the application process, averaged over all processors and all iterations, per iteration. Since the sender and receiver are synchronized for the nearest-neighbor vector messages, the measured receiving time is nearly the same as the actual communication time. Figure 17 shows the predicted and measured receiving times for the neighbor communication pattern. In this case the maximum error is about 15%. Since the one-word messages from inner product computations and coarse grid right-hand side broadcasts

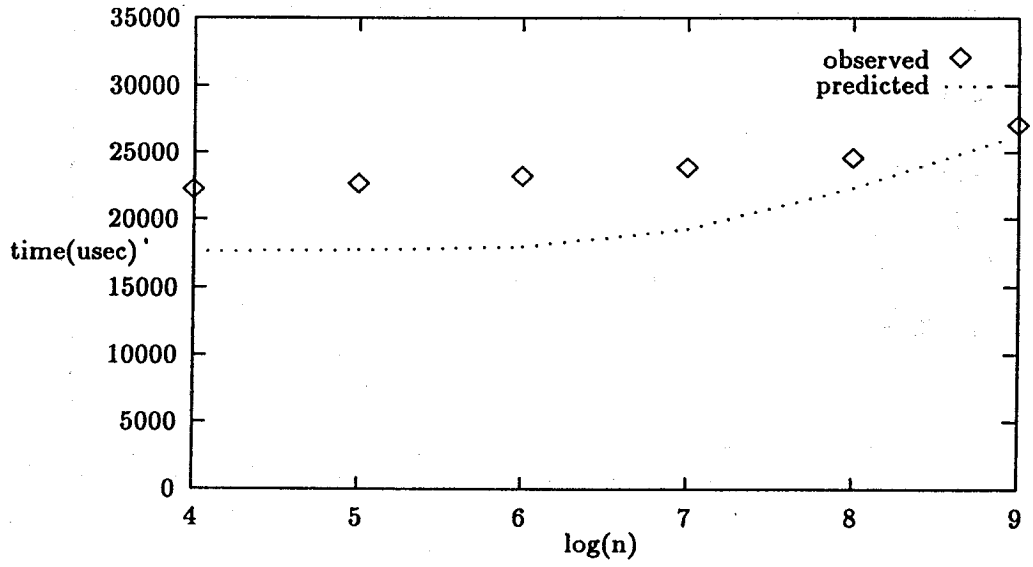


Figure 16: *Estimated sending time (T_s) versus experimentally measured sending time for the native DD code (without explicit synchronization), as a function of the log of the subdomain size. Note that, unlike Figures 11, 12, and 14, the vertical scale is linear in Figures 16,17, and 19.*

are omitted, a clear (linear in n) trend, induced by the domination of the transmission time over other fixed overheads, is visible.

5.2 A Multigrid Application

5.2.1 Algorithm

The second model application is transient simulation of incompressible Navier-Stokes flow in a two-dimensional square cavity filled with fluid, driven by an oscillatory rigid lid. The numerical method is based on a standard uniform grid spatial discretization and implicit time discretization for a velocity-pressure formulation of Navier-Stokes with a hybrid space-parallel/time-parallel multigrid solver. A multigrid solver uses a succession of grid presenting different refinements of the same problem, in order to iteratively damp the component of the error at each wavenumber on the grid for which its particular damping factor is most rapid, rather than damping all error components on the same grid. Space parallelism is achieved through domain partitioning, with one processor per subdomain, as in the first model application, though we permit both stripwise and boxwise decomposition in this case, in order to obtain more flexibility in the number of subdomains, while still preserving the uniformity of each subdomain. Time parallelism is achieved by assigning identically spatially decomposed time planes to disjoint sets of processors. The motivation for time parallelism is the degradation of efficiency in space parallelism that is due both to degrading perimeter-to-area (or surface-to-volume) ratios of conventional implicit methods, and to degrading convergence rate as global coupling is sacrificed in the MG "smoother," which is the error-reducing operation at the heart of MG, performed on a partition of a grid at a given refinement level. The effectiveness of time par-

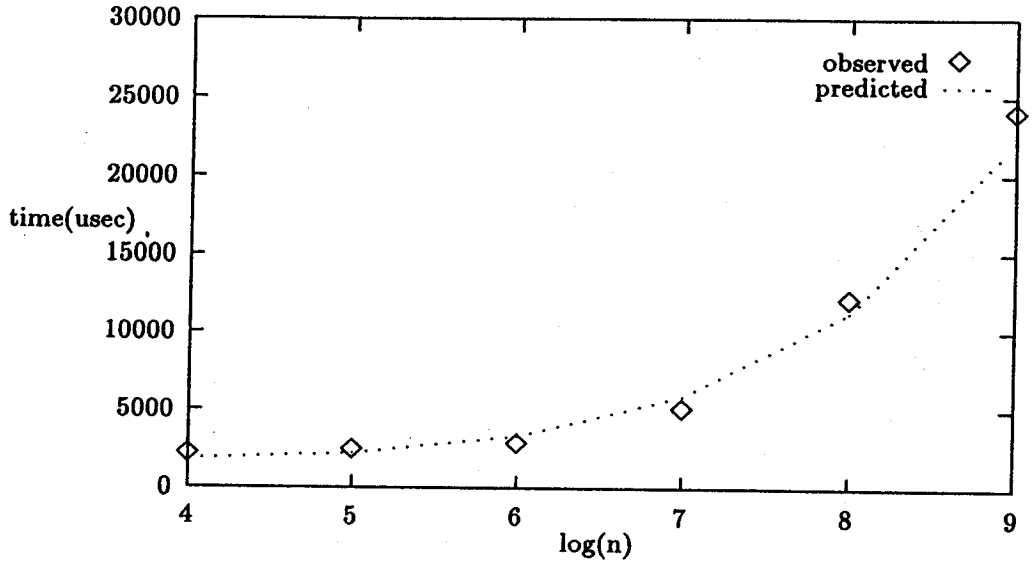


Figure 17: *Estimated receiving time (T_r) versus experimentally measured receiving time for neighbor communication pattern in the artificially synchronized version of the DD code, as a function of the log of the subdomain size.*

allelism is counter-intuitive because of causality. Nonetheless, it is more effective than space parallelism in many practical parameter ranges, when the time resolution of a transient flow is required.

In the limit of pure time parallelism, p processors work concurrently on p different time planes of the transient solution. In the limit of pure space parallelism, only one time plane is computed at a time. Multigrid is used in the spatial direction only; there is no time coarsening. (Time coarsening is worthy of attention in other contexts (see, e.g. [10]), but is irrelevant to our immediate purpose for this second application, namely to introduce a communication complexity that scales to the same asymptotic order in problem size as the computation complexity.)

A multigrid solver is defined by a grid coarsening strategy, a cycle for visiting successively coarsened grids, a smoother designed to reduce the highest wavenumber errors on a grid of a given level, and intergrid transfer operators to map the solution or its residual between grids at adjacent level. As with the DD application, it is beyond the scope of this paper to provide a self-contained specification of the MG algorithm. It should suffice to specify for *cognoscenti* that: the spatial coarsening is by powers of two in a simple V-cycle scheme, the semi-implicit method for pressure-linked equations (SIMPLE) defines the linearization, incomplete LU (ILU) decomposition the smoother, and standard full-weighting is used for intergrid transfers. The space parallelism enters through the elimination of certain off-diagonal blocks of the ILU factorization. The code was originally written for the Intel Hypercube by Horton [9].

The communication patterns and the amount of traffic vary with the allocation of available processors between space and time, as well as with the refinement of the spatial grid, with the result that in this second application a wide range of message sizes, message numbers

and message patterns are observed, depending on three factors: the number of physical time steps simultaneously solved for, the number of domain partitions, and the number of spatial coarsening levels. The most important observation about the computation and communication complexity, however, is that their asymptotic sizes are of equal order. Consider the purely time parallel limit of p planes of $n \times n$ gridpoints. Transferring the full plane of data between time levels is an $\mathcal{O}(n^2)$ operation, which is the same as the $\mathcal{O}(n^2)$ arithmetic complexity of the stencil operations of residual evaluation and ILU smoothing in the fine grid sweep of the MG algorithm.

5.2.2 Implementation

In Figure 18, we show the main patterns of communication generated between processors assigned to different time-steps ("in time") and between processors assigned to the same time step ("in space"). p_x is the number of grid partitions (p_x processors are assigned to solving the problem for every time step), while p_t is the number of consecutive time steps (p_t is the number of groups of p_x processors, each group operating on a different time step). Global operations are not shown in Figure 18; in general, their number is not constant, depending on the number of grid levels (kept fixed) and on the factorization of p into $p_t \times p_x$. The large messages are those carrying grid information (labeled G in the figure) between processors assigned to different time steps. The size of these messages decreases as p_x increases, for a constant number of time steps p_t, as the individual space domains are partitioned over more processors. For a given number of available processors, the size of the messages increases with p_t, as the space grid partitions become larger. The messages labeled "IR" and "IL" carry vectors of edge data right and left across spatial processor boundaries in the stripwise decomposition shown.

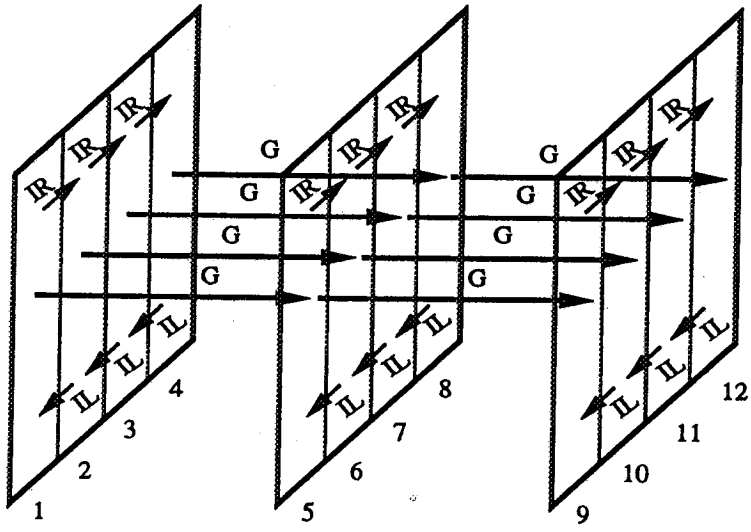


Figure 18: *Principal communication patterns for a single time step in the MG code, featuring both time ($p_t = 3$) and strip-based space ($p_x = 4$) parallelism for $p = 12$ processors.*

Figure 19 shows experimental versus predicted timings, for $p_x = 1$ and p_t between 2 and 12. There are two predicted curves: one for when all the communication operations are synchronized (and therefore the contention on the communication network is maximum), and one for the idealized case of no contention. For more than three workstation processors, the

network (as opposed to the processing overhead) is the communication bottleneck and thus the communication time increases linearly with the number of processors. On the other hand, for two processors the processing overhead represents the actual communication bottleneck and therefore the communication time does not increase at the same rate between two and three processors as in the other cases. This effect was anticipated at the end of section 4.

The measured communication time is bounded by the limits of the zero and maximal contention predicted curves. The difference between the maximal contention prediction and the actual communication time is due primarily to the lack of synchronization. The estimated communication times assume that all like messages are sent synchronously by all processors. In practice, the processors do not finish the computation phase at the same time and therefore the message sending is initiated at slightly staggered moments. This is due to slight workload imbalances and to nondeterministic factors that arise even when identical workstations have the same amount of work to perform. The computation time is influenced by the cache memory system, interrupt service, task switching and page swapping beyond the control of the application.

As in the DD examples, we modify the application so that, before sending, all processors are synchronized. The results obtained are also plotted against the predictions in Figure 19. In the synchronous case, the measured data are very close to the predictions (within 17%). Moreover, the difference should be even smaller if we could measure the real communication times (T_c) and not just the sending times (T_s). As a conclusion, the difference between the predicted communication time and the actual results expresses in some way the degree of the application synchronism. When the measured results are close to the synchronous predictions, the processors send messages at almost the same moments in time, which results in greater contention on the communication network and larger communication time.

In Figure 20 we consider different numbers of processors both in time and space. Since the main data traffic occurs between consecutive planes, we do not consider the processors in the last plane that only receive data. Between processors in the same plane a large number of small messages (several hundreds) are exchanged. This enforces a "natural" synchronization and, therefore, the time differences between the synchronized and non-synchronized (original) version of the application are smaller. This can be observed, especially, when processors in only one plane have to send data, i.e. $p_t = 2$. On the other hand, for $p_t = 4$ there are 3 planes that concurrently send data in time: plane 1 to plane 2, plane 2 to plane 3 and plane 3 to plane 4. Since the messages exchanged in the same plane synchronize only with processors in that plane, the processors in different planes are not so tightly synchronized and therefore the differences between the synchronized and non-synchronized version of the application are larger.

5.3 Discussion

Each of the two applications above gives rise to a small set of communication subprograms, such as global reduction or exchange of surface (resp., volume) data between spatially (resp., temporally) neighboring processors. These subprograms are called with message sizes ranging from one word to the order of the number of words of data in the problem. The hyperbolic model performs well for each communication subprogram class. It even has some value (see Figure 16) in predicting measurements averaged over all of the different communication subprograms in the algorithmically correct proportions.

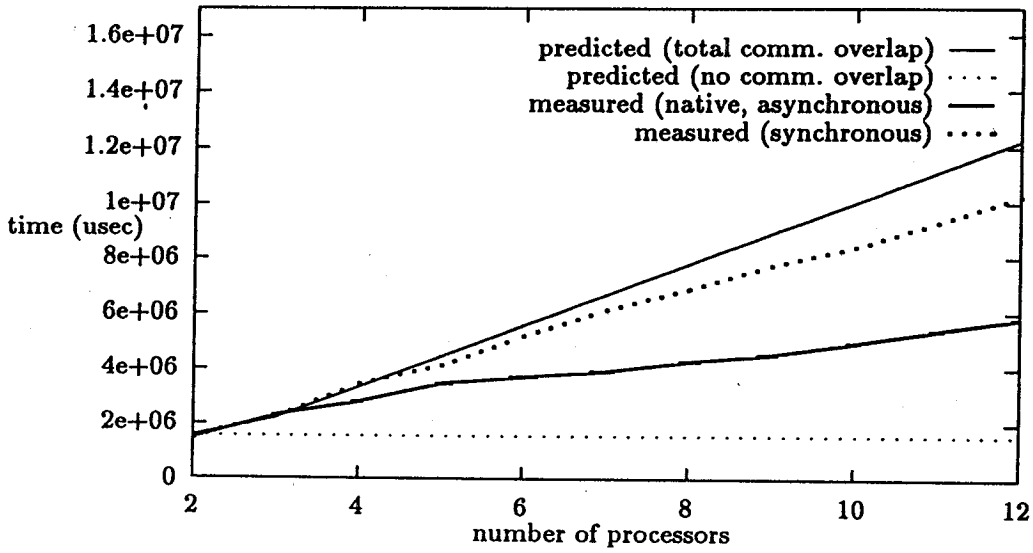


Figure 19: *Predicted sending times (T_s) for two extreme cases of zero contention on the communication network (no communication overlap) and maximum contention on the communication network (full communication overlap) versus the measured sending times for both the native MG code and the artificially synchronized version of the code. These results are for maximal time parallelism ($p_x = 1$ and $p_t = p$) which leads to the largest average message size.*

p_x	p_t	pred. (no ovlp.)	pred. (total ovlp.)	meas. (async)	meas. (sync)
2	2	925,480	1,374,963	1,137,562	1,264,402
2	4	925,480	3,162,490	2,110,917	2,616,590
4	2	509,125	1,343,780	1,158,536	1,273,106

Figure 20: *Predicted and measured sending times (in microseconds) for one multigrid V-cycle, for varying degrees of time and space parallelism, using either 4 or 8 processors.*

One of the applications (time-parallel multigrid) is limited by network contention, while the other application (domain decomposition) is limited only by irregularities in computation time and frequent synchronization. Both limitations are serious as regards scalability, particularly in cluster computing environments *without* dedicated nodes. Future algorithmic design should be heavily influenced by such communication analyses. In particular, the inner product operations used to drive the conjugate gradient iterations are particularly burdensome and their synchronization cost should be reduced by algorithmic variants that block several consecutive iterations into one set of global reduction operations. Indeed, the synchronization costs of conjugate-gradient-type methods may lead to a resurgence of interest in Chebyshev-like methods.

The modifications made to the original applications programs to produce artificial synchrony are for purposes of demonstrating the ability of the hyperbolic model to predict contention, only, and are not recommended in production versions.

Tests on architectures other than Ethernet Sparcstation clusters, with message-passing protocols other than p4, using applications other than domain decomposition and time-parallel multigrid are planned, to further define the range of applicability of the hyperbolic model.

6 The LogP Model

Recently, a new model of parallel computational complexity for massively parallel processors, called LogP, has been developed at Berkeley [4]. 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 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 - the upper bound for the delay of a message transmission between the source and destination processors.

2. o - the time interval required to send or receive a message. During this time the processor cannot perform any other operations.

3. g - the *gap*, defined as the minimum time interval between two consecutive message transmissions or receptions.

4. P - the number of modules.

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 21). Since for $n = 1$ the second expression is reduced to the first, we consider further only the second one.

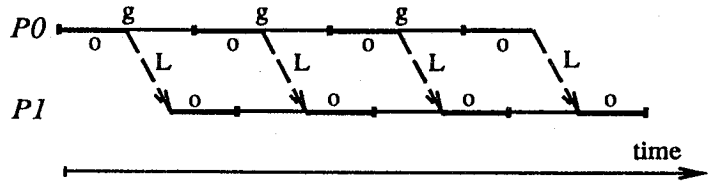


Figure 21: *The time diagram for sending 4 consecutive messages, from P0 to P1, in the LogP model. Here, $g \geq o$.*

To capture the LogP parameters in the hyperbolic model we use the communication graph from Figure 9, 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 are of the same size, we take the size of the data unit and the packet size to be the same (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 (15).

When we write $x \to 0$ 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, $x \to 0$ 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:

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

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:

$$\frac{3}{4} \leq \frac{T_{hyp}(n)}{T_{LogP}(n)} \leq 1. \tag{18}$$

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 21) $T_{s\_LogP}(n) = T_{r\_LogP}(n) = no$. 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 sending and receiving primitives of the hyperbolic model are *preemptive*. From Figure 8 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 $n \geq 1$:

$$\frac{3}{4} \leq \frac{T_{s\_hyp}(n)}{T_{s\_LogP}(n)} \leq 1. \tag{19}$$

7 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. Because of the way its parameters are fit to experiments, the model captures both small-message and large-message timing behavior well. 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 set 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 hyperbolic model is tractable.

In the limit of small uniform messages that affords direct comparison with the state-of-the-art LogP model, appropriate for tightly coupled architectures, the hyperbolic and LogP models predict the same timings for elementary communication operations to within a factor of 3/4.

The model can be used to provide insight into communication performance of actual distributed scientific applications. A domain decomposition code for solving elliptic PDEs and a time-parallel multigrid method for transient simulation of Navier-Stokes cavity flow are chosen for demonstration purposes, because of their different synchronization/communication ratios and complementary communication patterns. Complementary bottlenecks to scalability are thus identified. Realistic analyses of communication such as these can be used to influence algorithmic design, for a given architecture, and vice versa.

8 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 Prof. Chet Grosch of Old Dominion University, and from a key working session with Prof.

Roger Hockney while a visitor at Old Dominion. Two of the authors of the LogP model have been helpful in clarifying aspects of our comparisons in Section 6 (though these comparisons should not be presumed definitive, insofar as the LogP authors are concerned). Finally, the generosity of Prof. Graham Horton of the University of Erlangen in donating his code, as well as his time in explaining the time-parallel MG algorithm, is worthy of special acknowledgement!

References

[1] J. H. Bramble, J. E. Pasciak and A. H. Schatz, *The construction of preconditioners for elliptic problems by substructuring, I*, Math. Comp., 47:103–134, 1986.

[2] R. Butler and E. Lusk, *Monitors, Messages, and Clusters: The p4 Parallel Programming System*, Argonne National Laboratory MCS Div. preprint P362-0493, and J. Parallel Comput., to appear.

[3] D. D. Clark, Van Jacobson, J. Romkey, H. Salwen, *An Analysis of TCP Processing Overhead*, IEEE Communications, pp. 23–29, June 1989.

[4] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian and T. von Eicken, *LogP: Towards a Realistic Model of Parallel Computation*, SIGPLAN Notices, **28**:1–12, 1993.

[5] R. Cypher and E. Leu, *The semantics of blocking and nonblocking send and receive primitives*, in Proceedings of the International Parallel Processing Symposium '94, Cancun, Mexico, IEEE Press, Los Alamitos, CA, pp. 729–735, 1994.

[6] S. Fortune and J. Wyllie, *Parallelism in Random Access Machines*, in Proceedings of the 10th Annual Symposium on Theory of Computing, San Diego, CA, 1978, ACM Press, New York, pp. 114–118, 1978.

[7] W. D. Gropp and D. E. Keyes, *Domain Decomposition on Parallel Computers*, Impact of Computing in Science and Engineering, **1**:421–439, 1989.

[8] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach*, Morgan Kauffman, San Mateo, CA, 1990.

[9] G. Horton, *Time-parallelism for the massively parallel solution of parabolic PDEs*, in Applications of High Performance Computers in Science and Engineering, Computational Mechanics Publications, Southampton (UK), December 1994 (to appear).

[10] G. Horton and S. Vandewalle, *A Space-Time Multigrid Method for Parabolic PDEs*, Report 6/93, IMMD 3, Universitaet Erlangen, July 1993 (to appear in SIAM J. Sci. Comput.).

[11] D. E. Keyes, *Domain Decomposition: A Bridge Between Nature and Parallel Computers*, in "Adaptive, Multilevel and Hierarchical Computational Strategies" (A. K. Noor, ed.), ASME, New York, pp. 293–334, 1992.

[12] D. E. Keyes and W. D. Gropp, *A Comparison of Domain Decomposition Techniques for Elliptic Partial Equations and their Parallel Implementation*, SIAM J. Sci. Stat. Comput., **8**:s166–s202, 1987.

[13] S. T. Leutenegger and X.-H. Sun, *Distributed Computing Feasibility in a Non-dedicated Homogeneous Distributed System*, Proc. of Supercomputing'93, Portland, OR, IEEE Computer Society Press, pp. 143-152, 1993.

[14] K. Maly, S. Khanna, E. C. Foudriat, C. M. Overstreet, R. Mukkamala, M. Zubair, R. Yerraballi, D. Sudheer, *Parallel Communications: An Experimental Study*, TR-93-20 Dept. of Comp. Sci., Old Dominion Univ., 1993.

[15] K. K. Ramakrishnan, *Performance studies in designing Network Interfaces: A Case Study*, in Proceedings of the 4th IFIP Conference on High Performance Networking '92, (A Danthine, O. Spaniol, eds.). Int. Fed. for Information Processing, pp.F3-1 – F3-15, 1992.

[16] W. Stallings, *Data and Computer Communications*, Macmillan, New York, 1991.

[17] W. R. Stevens, *TCP/IP Illustrated, Volume I: The Protocols*, Addison-Wesley, Reading (MA), 1994.

[18] L. H. Turcotte, *A Survey of Software Environments for Exploiting Networked Computing Resources*, Technical Report, Engineering Res. Ctr. for Computational Field Simulation, Mississippi State Univ., June, 1993.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
-	September 1994	Contractor Report

4. TITLE AND SUBTITLE
A SIMPLE HYPERBOLIC MODEL FOR COMMUNICATION IN PARALLEL PROCESSING ENVIRONMENT

5. FUNDING NUMBERS
C NAS1-19480
WU 505-90-52-01

6. AUTHOR(S)
Ion Stoica
Florin Sultan
David Keyes

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science
and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 94-78

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-194984
ICASE Report No. 94-78

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Michael F. Card
Final Report
To be submitted to Journal of Parallel and Distributed Computing

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited

Subject Category 60, 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

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. Dedicated interprocessor links parameterized by a latency and a transfer rate that are independent of load are assumed by many existing communication models; such models are unrealistic for workstation networks. The communication system is modeled as a directed communication graph in which terminal nodes represent the application processes that initiate the sending and receiving of the information and in which internal nodes, called communication blocks (*CB*s), reflect the layered structure of the underlying communication architecture. The direction of graph edges specifies the flow of the information carried through messages. Each *CB* is characterized by a two-parameter hyperbolic function of the message size that represents the service time needed for processing the message. The parameters are evaluated in the limits of very large and very small messages. Rules are given for reducing a communication graph consisting of many to an equivalent two-parameter form, while maintaining an approximation for the service time that is exact in both large and small limits. The model is validated on a dedicated Ethernet network of workstations by experiments with communication subprograms arising in scientific applications, for which a tight fit of the model predictions with actual measurements of the communication and synchronization time between end processes is demonstrated. The model is then used to evaluate the performance of two simple parallel scientific applications from partial differential equations: domain decomposition and time-parallel multigrid. In an appropriate limit, we also show the compatibility of the hyperbolic model with the recently proposed LogP model.

14. SUBJECT TERMS
communication modeling, parallel processing for partial differential equations, cluster computing, workstation networks

15. NUMBER OF PAGES
41

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		

NSN 7540-01-280-5500

Standard Form 298(Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

☆U.S. GOVERNMENT PRINTING OFFICE: 1994 - 628-064/23062