Modeling the Communication Performance of the IBM SP2*

Gheith A. Abandah  Edward S. Davidson
Advanced Computer Architecture Laboratory, Department of EECS
University of Michigan
1301 Beal Avenue, Ann Arbor, MI 48109–2122
gabandah.davidson@eecs.umich.edu

Abstract

The objective of this paper is to develop models that characterize the communication performance of a message-passing multicomputer by taking the IBM SP2 as a case study. The paper evaluates and models the three aspects of the communication performance: scheduling overhead, message-passing time, and synchronization overhead. Performance models are developed for the basic communication patterns, enabling the estimation of the communication times of a message-passing application. Such estimates facilitate activities such as application tuning, selection of the best available implementation technique, and performance comparisons among different multicomputers.

1. Introduction

A distributed-memory multicomputer consists of multiple processor nodes interconnected by a message-passing network. Each processor node is an autonomous computer consisting of a central processing unit (CPU), memory, communication adapter, and—for at least some nodes—mass storage and I/O devices.

Figure 1 shows a general model for a message-passing multicomputer. The interconnection network provides the communication channels through which the processor nodes exchange data and coordinate their work in solving a parallel application. Different types of interconnection networks vary in topology and throughput; meshes, hypercubes and multistage interconnection networks (MINs) are three of the commonly used topologies. The communication adapter provides the interface between the processor node and the interconnection network. While some simple communication adapters are directly controlled by the CPU, other more sophisticated adapters do communication processing, error checking and correction, and direct memory access (DMA). The number of communication links provided by each communication adapter depends on the topology of the interconnection network; the communication adapter usually has one communication link in a MIN, \( \log_2 P \) links in a hypercube with \( P \) processor nodes, and \( 2D \) links in a D-dimensional mesh.

In order to develop efficient message-passing applications, programmers need good performance models for a wide variety of communication patterns. The available models for message-passing multicomputers, however, either do not cover all the important communication patterns or do not give sufficient information for direct use in tuning a message-passing application. This paper and a similar independent effort [1] address this need.

Most of the work in developing performance models for message-passing multicomputers is centered around benchmarks. The NAS Parallel Benchmarks (NPB) [2] were developed to study the performance of parallel supercomputers, they consist of five parallel kernels and three simulated applications. Together, they mimic the computation and data movement characteristics of large-scale computational fluid dynamics applications. Another suite is PARKBENCH [3] which contains low-level benchmarks for measuring basic computer characteristics, kernels to test typical scientific subroutines, and compact applications to test complete problems.

Benchmarks like NPB are useful for comparing different machines, but they do not separate computation performance from communication performance. PARKBENCH has benchmarks for some of the communication patterns,

---

* The University of Michigan Center for Parallel Computing, site of the SP2, is partially funded by NSF grant CDA–92–14296. G. Abandah is partially funded by Fulbright Fellowship.
2. IBM SP2

The IBM Scalable POWERparallel SP2 [6] connects 2 to 512 RISC System/6000 POWER2 processors [7] via a communication subsystem. This subsystem is based upon a low-latency, high-bandwidth interconnection network called the High Performance Switch (HPS). The HPS is a bidirectional MIN, where each communication link has two channels which carry data in opposite directions. The HPS scales the bisection bandwidth linearly with the number of nodes while maintaining a fixed number of communication ports per processor node.

Each SP2 node is incorporated into a logical frame (see Figure 2), which encompasses up to 16 nodes connected to one side of the switch board. The switch board has two stages; each stage has four switching elements, the first is connected to the processor nodes, and the second is used to connect with other frames. The HPS is constructed from 4-way to 4-way bidirectional switching elements. The switching elements—physically 8-input, 8-output devices—can forward packets from any input port to any of the 8 output ports as directed by the route information.

Nodes send messages to other nodes by breaking the messages into packets and injecting these packets into the HPS. Each packet contains route information that is used by the switching elements to forward the packet correctly to its destination. The smallest unit on which flow control is performed is called a flit, which is one byte in the SP2. The width of each link is also one byte. Each packet has one length flit followed by one or more route flits then the data flits; total packet length is at most 255 flits.

The HPS flow control method is buffered wormhole routing. Each flit of a packet is advanced to the appropriate output port as soon as it arrives at a switching element input port. When the head of a packet is blocked, the flits are buffered in place. As soon as the output port is free, packet transfer resumes.

The switching element of the board is the Vulcan switch chip [8]. The HPS operates at 40 MHz, providing a peak bandwidth of 40 MB/sec over each of the two channels of each communication link. The Vulcan switch chip contains 8 receiver modules, 8 transmitter modules, an 8 by 8 crossbar, and a central queue for buffering the blocked flits. In the absence of contention, packet flits incur 5 cycles of latency while cutting through the chip via the crossbar path.

An SP2 processor node is attached to the HPS via the En-
hanced Communication Adapter. This adapter incorporates an Intel i860 XR 64-bit microprocessor, 2 KB input FIFO (first-in-first-out buffer), 2 KB output FIFO, and two DMA engines. The communication adapter does communication coprocesing, data checking, and DMA between the memory and the two FIFOs.

Three types of processor nodes are available for the SP2; Thin, Thin2, and Wide nodes. These three types of nodes differ in the data cache size, memory bus width, and the bus width between the data cache and the Floating Point Unit (FPU). The experiments of this paper were carried out on the University of Michigan Center for Parallel Computing SP2 which has 32 POWER2 Thin nodes with a 15 nsec clock. Each node has a 32 KB instruction cache, 64 KB data cache, 64 bit memory bus, and a 128 bit bus between the data cache and the FPU (a 2-port bus). All the experiments were carried out during exclusive reservation (no processes running for other users). The measured data show low standard deviation, usually only a few percent of the measured time.

The SP2 supports the MPI [9], PMVe [10], and MPL [11] message-passing libraries. MPI is the emerging Message Passing Interface standard, PMVe is IBM’s implementation of the popular PVM [12], and MPL is IBM’s proprietary Message Passing Library. MPL is similar to MPI and contains Fortran and C subroutines that allow managing and querying the task environment, sending and receiving messages, performing collective communication, and implementing barrier synchronization.

MPL has two types of sending and receiving subroutines; blocking subroutines, e.g. MP_BSEND and MP_BRECV, and non-blocking subroutines, e.g. MP_SEND and MP_RECV. Blocking subroutines wait until application buffers are free for reuse before returning. Non-blocking subroutines, on the other hand, return as soon as possible, with communication processing continuing regardless of the state of the application buffers. For Thin nodes, the performance of the non-blocking subroutines is essentially the same as the blocking routines.

MPL has two implementations [13]; the Internet Protocol implementation (IP) which allows shared usage of the processor nodes, and the User Space implementation (US) which has better performance, but has a limit of one parallel task per processor node. Unless otherwise stated, our experiments were done using the US implementation of the MPL message-passing library.

3. Scheduling time

Scheduling time is the time needed by the parallel environment to start and finish parallel tasks on the processor nodes. This time includes the overhead of allocating processor nodes for the parallel task, distributing the task executable code, starting the task on the processor nodes, and freeing the allocated processor nodes at the task completion.

Since the SP2 supports only static scheduling—no scheduling of new nodes is possible during run time—the scheduling time is incurred once and is significant for short programs. The scheduling time has a small effect on the scalability of long programs, but limits the scalability of short programs. To evaluate this time, we run a simple program that prints the task id on a varying number of processors, and we measure the execution wall time. Figure 3 shows the scheduling time for 10 runs.

We have two observations on the results of these runs:

1. The SP2 scheduling time changes from one run to another with a wide variance; ranging from 2.8 sec to 4.8 sec for one processor and 57.7 sec to 538.6 sec for 32 processors.

2. The scheduling time increases dramatically as the number of processors increases; the average scheduling time for 32 processors is 234 sec.

The SP2 has a parallel task server called the Resource Manager that runs on one of the SP2 nodes and manages the allocation of nodes to parallel tasks. When a parallel task is initiated at one node, the node connects to the Resource Manager and requests a list of nodes that can be used. The requesting node then loads a copy of the executable code on each of the allocated nodes and starts the task. The requesting node is informed when each node completes and takes over to terminate the task and inform the Resource Manager to free the nodes. Apparently, this scheduling process for tens of nodes generates heavy traffic and requires some bookkeeping that takes highly variable and sometimes long periods of time.

4. Message-passing time

Message-passing time is the most important communication time for long applications with heavy message passing. Estimating the time needed to carry out a communi-
cation process for data transfer in a message-passing multi-
computer is a complex problem. The message-passing time
depends on the characteristics of the processor nodes and the
interconnection network, the message-passing library used,
the communication pattern, the message size, the number
of processor nodes, and the distance between the processor
nodes.

Our approach to developing communication perfor-
mance models for a particular message-passing library
of a given multicomputer is to measure and model the
performance of a set of common communication patterns.
This set contains the basic communication patterns that are
selected in such a way that other complex communication
patterns can be constructed from these basic patterns.
Hence, the time of a complex communication pattern is
estimated by summing the times of its basic components.
The basic communication patterns are Point-to-point, Ex-
change, One-to-many, Many-to-one, and Many-to-many. In
addition to the basic communication patterns, we model the
performance of some of the MPL collective communication
subroutines (subroutines that perform communication op-
erations by a group of processor nodes), namely Broadcast
and Combine which are efficient and frequently used in
message-passing applications. Although MPL also has
collective communication subroutines for One-to-many,
Many-to-one, and Many-to-many communication patterns,
we implement these patterns using simple send and receive
subroutines. Our implementations for these patterns are
less restrictive than the MPL subroutines, but have similar
performance.

For each experiment, we have prepared a Fortran pro-
gram that calls the MPL message-passing subroutines. Each
program performs one communication pattern many times,
and reports the minimum time, the average time, and the
standard deviation. These programs are executed with varying
numbers of processors \(p\), message lengths \(n\), and
internode distances. The timing data gathered from executing
these programs is analyzed and used to develop formulas,
via curve fitting, that give the time as a function of the com-
munication variables.

The SP2 communication latencies vary by only a few per-
cent as the internode distance is varied. We have evaluated
this small effect (see Figure 6), but for simplicity we do not
include it in the SP2 communication models. In the follow-
ing subsections, we describe the communication patterns,
describe the experiments used to gather their timing data,
and develop a performance model for each pattern.

### 4.1. Point-to-point communication

The Point-to-point communication experiment measures
the basic communication properties of a message-passing
multicomputer. In this experiment a message is sent from
processor A to processor B. Processor B receives the mes-
sage and immediately returns it back to processor A. This
pattern is repeated many times and the minimum round trip
time is picked, as it reflects the least extraneous system inter-
ference. The reported Point-to-point communication time
is one half the minimum round trip time. The Point-to-
point communication time depends on three (possibly over-
lapping) times: transmission, propagation, and reception
times. This experiment was implemented by making one
processor call \texttt{MP\_BSEND} then \texttt{MP\_BRECV}, and the other
call \texttt{MP\_BRECV} then \texttt{MP\_BSEND}.

Figure 4 shows the minimum Point-to-point communica-
tion time on the SP2 using US and IP. For US, the latency of
short messages approaches 47 \(\mu\text{sec}\) (there is a discontinu-
ity at \(n = 0\) bytes where the latency is only 43 \(\mu\text{sec}\)), and
the transfer rate of large messages reaches 35.6 MB/sec (see
Figure 5). The transfer rate is found by dividing the mes-
sage length by the minimum communication time. For IP,
the latency of short messages approaches 329 \(\mu\text{sec}\), and the
transfer rate of large messages reaches 10.6 MB/sec.

More detailed experiments were done to learn more about
the Point-to-point communication time characteristics. The
experiment described above was repeated for every message
length in the range 0 to 2000 Bytes; the results are shown in Figure 6. This was done twice, first with two nodes connected to the same switching element (Near nodes) and second with two nodes from different frames (Far nodes).

The results show that the Point-to-point communication time is about 2.5 μsec longer when the communication is between Far nodes. Less than 10% of this time is due to the latency in traveling through more switching elements (3 elements × 5 cycles × 15 nsec/cycle), the rest is due to the processor overhead of handling more complex route information. The results also show periodic discontinuities in the communication time due to packetization. The discontinuities are at 217 + 232i Byte message lengths; where \( i \in \{0, 1, 2, \ldots\} \). This implies that the first packet has 216 data flits and 39 overhead flits, and the subsequent packets have 232 data flits and 23 overhead flits. The effective HPS bandwidth is thus \( 40 \times 232 / 255 = 36.4 \text{ MB/sec} \).

The Point-to-point communication time often can be expressed by a few simple parameters [14]: \( r_\infty \) is the asymptotic transfer rate in MB/sec, and \( t_0 \) is the asymptotic zero message length latency in μsec. The time (in μsec) for Point-to-point communication as a function of \( n \), \( T_{pp}(n) \), is given by the following equation:

\[
T_{pp}(n) = t_0 + \frac{n}{r_\infty}
\]

For small messages, the setup time \( t_0 \) is dominant, while for large messages the transfer time governed by \( r_\infty \) is dominant. We have observed that this simple model is not accurate for a wide range of \( n \); it has about 30% relative error at some points. This suggests splitting the \( n \) domain into regions, where each region has its own \( r_\infty \) and \( t_0 \). Splitting the model into carefully chosen regions enabled us to get models that are more accurate with relative error never exceeding 5%. The parameters and regions for the Point-to-point communication time using US are:

\[
t_0 = \begin{cases} 
47 & n < 217 \\
55 & 217 \leq n \leq 2K \\
74 & 2K < n \leq 64K \\
399 & n \geq 64K
\end{cases}
\]

\[
r_\infty = \begin{cases} 
23.5 & n < 217 \\
22.6 & 217 \leq n \leq 2K \\
29.3 & 2K < n \leq 64K \\
36.2 & n \geq 64K
\end{cases}
\]

The boundary between the first and the second regions is due to packetization. The boundary between the second and the third regions is due to the size of the communication adapter’s FIFO buffer (2 KB). The boundary between the third and fourth regions is due to the fact that MPL does fewer data movements between the communication buffers for large messages.

The Point-to-point communication time using IP can be modeled by the following parameters and regions:

\[
t_0 = \begin{cases} 
329 & n \leq 8K \\
460 & n > 8K
\end{cases}
\]

\[
r_\infty = \begin{cases} 
11.2 & n \leq 8K \\
10.7 & n > 8K
\end{cases}
\]

These parameters show that the US implementation has lower latency and its transfer rate is about 3.4 times higher than the IP implementation. Despite its lower performance, IP can be used with applications that have poor load balance to get higher processor utilization by running multiple tasks on each processor node (only one US task per node is allowed).

Finding a model’s regions can be done automatically using a simple algorithm. The algorithm starts with a region containing the first two points and incrementally adds points to the region. Whenever a new point is added, curve fitting is used to find the region parameters which are used to check the model accuracy. If the model does not meet a preset error margin, the algorithm removes the last point and uses it as the first point in a new region.

### 4.2. Exchange communication

This experiment measures the performance of the processor when it is using its communication link in both directions simultaneously. In this experiment we have two processors; each processor sends a message to the other processor and receives the message sent to it. The time needed to complete receiving one message and sending one message is the Exchange communication time. This experiment was implemented by making the two processors repeatedly call the subroutine MP_BSENDRECV.

Figure 4 shows the minimum Exchange communication time on the SP2, and Figure 5 shows the transfer rate. The Exchange time for small messages approaches 41 μsec, and the transfer rate for 1 MB messages is 41.2 MB/sec. The transfer rate is found by dividing twice the message length by the minimum Exchange communication time. The Exchange communication time model is similar to the Point-
to-point model:

\[ T_{\text{exch}}(n) = t_0 + 2n \frac{n}{r_{\infty}} \]

The parameters and regions for the Exchange communication time are:

\[ \begin{align*}
  t_0 &= \begin{cases} 
    40 & n < 4\text{K} \\
    -52 & 4\text{K} \leq n \leq 16\text{K} \\
    167 & n > 16\text{K} 
  \end{cases} \\
  r_{\infty} &= \begin{cases} 
    56.8 & n < 4\text{K} \\
    32.4 & 4\text{K} \leq n \leq 16\text{K} \\
    41.4 & n > 16\text{K} 
  \end{cases}
\]

In the Exchange communication, the transfer rate rises to 40.8 MB/sec for 4 KB messages, then drops significantly as the message length grows from 4 KB to 16 KB, and finally recovers asymptotically to 41.2 MB/s. From this data, we conclude that the performance can sometimes be improved if long messages are divided into a series of 4 KB chunks.

When using the communication link in both directions as in the Exchange communication, we get a higher transfer rate than when using it in one direction only as in the Point-to-point communication. For example, although the rate does not approach the 2× factor, the Exchange transfer rate is 17% higher than the Point-to-point transfer rate for a 1 MB message.

### 4.3. One-to-many communication

This experiment measures the outbound node performance of a message-passing multicomputer. The sender processor sends a distinct message to \( p \) different processors. The time taken by the sender processor to finish sending these messages is the One-to-many communication time. This experiment was implemented by making the sender processor call the subroutine \( \text{MP\_SEND} \) \( p \) times, then call \( \text{MP\_WAIT} \) to ensure that all of the sends have completed; each receiver calls \( \text{MP\_BRECV} \). Figure 7 shows the average One-to-many communication time on the SP2. This data shows that the One-to-many transfer time is proportional to \( n \) and to \( p \), and the latency is linear in \( p \).

![Figure 7. One-to-many communication: sender time.](image)

For the One-to-many communication, and the rest of the collective communication patterns described in the following subsections, we model them as functions of \( n \) and \( p \), \( T_{\text{comm}}(n, p) \),

\[ T_{\text{comm}}(n, p) = t_{\text{comm}}(p) + \pi_{\text{comm}}(p)n \]

where \( t_{\text{comm}}(p) \) is the setup time, and \( \pi_{\text{comm}}(p) \) is the transfer time per byte. To develop such performance models we use two-stage curve fitting; first we find \( t_{\text{comm}} \) and \( \pi_{\text{comm}} \) for each \( p \), and second we perform curve fitting to find \( t_{\text{comm}}(p) \) and \( \pi_{\text{comm}}(p) \). The resulting model for the One-to-many communication time is:

\[ T_{1m}(n, p) = t_{1m}(p) + \pi_{1m}(p)n \]

\[ = (-5.5 + 15.5p) + (0.031p)n \]

This model implies that the setup time is 10 \( \mu \text{sec} \) for a single destination, and 15.5 \( \mu \text{sec} \) for each additional destination, apparently due to the added overhead of managing more than one ongoing message. As \( n \) increases toward 64 KB, the overall transfer rate \( (pn/T_{1m}) \) goes toward 32 MB/sec. Figure 7 shows the model and the measured data (within 20% for \( p \geq 4 \)). The model does show a loose fit for some points, particularly at the knee for small \( p \). The model can be made more accurate by splitting it into several regions as done in the Point-to-point communication model.

### 4.4. Many-to-one communication

This experiment measures the inbound node performance of a message-passing multicomputer. The receiver processor receives a distinct message from \( p \) different processors. The time taken by the receiver processor to finish receiving these messages is the Many-to-one communication time. This experiment was implemented by making the receiver processor call the subroutine \( \text{MP\_RECV} \) \( p \) times, then call \( \text{MP\_WAIT} \) to ensure that all of the receives have completed, each sender calls \( \text{MP\_BSEND} \). Figure 8 shows the average Many-to-one communication time on the SP2. Once again
the Many-to-one transfer time is proportional to \( n \) and to \( p \), and the latency is linear in \( p \).

The model for the Many-to-one communication time is:
\[
T_{m1}(n, p) = t_{m1}(p) + \tau_{m1}(p)n = (3 + 13.3p) + (0.0285p)n
\]
which is within 20% relative error. As \( n \) increases toward 64 KB, the overall transfer rate goes toward 35 MB/sec.

4.5. Many-to-many communication

This experiment measures the total saturation bandwidth of a message-passing multicomputer, and finds how this bandwidth scales with the number of processors. In this experiment each processor sends different messages of length \( n \) to the other \( p - 1 \) processors, and receives \( p - 1 \) messages. The time needed by a processor to send and receive its share of messages is the Many-to-many communication time. This experiment was implemented by making every processor call \( \text{MP}_\text{SEND} \) once for each other processor, then call \( \text{MP}_\text{_RECV} \) for each other processor, and finally call \( \text{MP}_\text{WAIT} \). Figure 9 shows the average Many-to-many communication time.

\[
T_{mm}(n, p) = t_{mm}(p) + \tau_{mm}(p)n = (3 + 40(p - 1)) + (-0.005 + 0.062(p - 1))n
\]
which is within 20% relative error. For large messages, the overall bidirectional rate for a processor \( \approx 32 \) MB/sec, implying good communication scalability even when the HPS is under the heaviest possible traffic with all the 32 processors actively sending and receiving messages.

4.6. Broadcast

This experiment measures the Broadcast performance of a message-passing multicomputer. Unlike One-to-many, the sender processor sends the same message to \( p \) different processors. The time taken by the sender processor to finish sending these messages is the Broadcast time. This experiment was implemented by making every processor call \( \text{MP}_\text{BCAST} \). Figure 10 shows the average Broadcast time on the SP2.

The model for the Broadcast time is:
\[
T_{bc}(n, p) = t_{bc}(p) + \tau_{bc}(p)n = \begin{cases} 
(9.6 + 14D) + (0.0083 + 0.015D)n & n < 217 \\
(6 + 12D) + (0.025 + 0.026D)n & n \geq 217 
\end{cases}
\]
where \( D = \lfloor \log_2 p \rfloor \). The logarithm terms in this model imply that the MPL library uses an efficient algorithm for implementing the broadcast.

4.7. Combine

This experiment measures the Combine performance of a message-passing multicomputer. In Combine, corresponding elements of a length \( n \) message from each of \( p \) processors, are reduced to one length \( n \) message which is sent to all \( p \) processors. This experiment was implemented by making every processor call \( \text{MP}_\text{COMBINE} \); the reduction operation used in this experiment is a double-precision summation. Figure 11 shows the average Combine time on the SP2.

The model for the Combine time is:
\[
T_{cb}(n, p) = t_{cb}(p) + \tau_{cb}(p)n = \begin{cases} 
(97D) + (0.11D)n & n < 217 \\
(114D) + (0.12D)n & 217 \leq n \leq 2K \\
(-50 + 191D) + (0.09D)n & n > 2K 
\end{cases}
\]
where \( D = \lfloor \log_2 p \rfloor \). Again the logarithm terms imply that the MPL library uses an efficient algorithm for implementing the combine.
5. Synchronization time

Synchronization between processor nodes in a message-passing multicomputer occurs implicitly when they transfer data. Figure 12 shows two processors engaged in a parallel task with phases of computation and communication. Processor P1 finishes its first computation phase before processor P2, but due to the communication phase where it exchanges data with processor P2, P1 waits until P2 is ready to begin the exchange, resulting in an implicit synchronization. After the exchange, the second computation phase begins on both processors.

MPL does have a subroutine for explicit synchronization (MP\_SYNC), which implements a synchronization barrier. We have done two experiments to evaluate the overhead of this subroutine. First we measured the average time to call this subroutine when all the processors enter the barrier simultaneously. This experiment was implemented by making every processor call the subroutine MP\_SYNC for many iterations. Figure 13 shows the average synchronization time for a varying number of processors.

Second we measured the minimum time to complete the synchronization barrier for the last entering processor (last-in). This experiment was implemented by making one processor do a long dummy computation before calling MP\_SYNC while the other processors call MP\_SYNC iteratively without any intervening computations. Figure 13 also shows the minimum last-in synchronization time for a varying number of processors.

The models for the synchronization times give the time as a function of the number of processors involved in the synchronization barrier. The average synchronization time $T_{\text{sync}}(p)$, and the minimum synchronization time for the last-in processor $T_{\text{last-in}}(p)$ can be approximated by the following models:

$$T_{\text{sync}}(p) = 84 \log_2 p$$

$$T_{\text{last-in}}(p) = 37 \log_2 p$$

These models imply that the MPL library uses an efficient implementation for the barrier. Logarithmic time can be achieved when the barrier is implemented using a distributed algorithm like the binary-tree method.

6. Conclusions and use of the models

In this paper we have presented an experimental method for systematically measuring the communication performance of a message-passing multicomputer and then modeling it analytically via simple curve fitting. We illustrated this method by carrying out a case study of the three aspects of the IBM SP2 communication performance: the scheduling overhead, the message-passing time, and the synchronization overhead.

The scheduling overhead of the SP2 varies greatly from one run to another, and increases as the number of scheduled processors increases. The high scheduling time for tens of processors limits the scalability of short-running programs.

We have identified the factors that affect the message-passing time and developed models for the common communication patterns. These models give the time as a function of the message length and the number of processors, and have two components; setup time, and transfer time:

$$T_{\text{comm}}(n, p) = t_{\text{comm}}(p) + \tau_{\text{comm}}(p)n$$

Modeling the communication time with just these two components gives simple and intuitive equations which are fairly accurate. Occasionally, several separate regions of $n$ are required, as determined from the observed data. System
implementation causes for these discontinuities have been found in some cases.

The message-passing time models for the SP2 reveal that it sustains an overall transfer rate better than 32 MB/sec for large messages using the basic communication patterns, and that the setup time scales linearly with \( n \). The MPL Broadcast and Combine collective communication subroutines use efficient distributed algorithms that result in setup and transfer times that are logarithmic in \( p \). We have seen no saturation effects as \( p \) increases up to 32 processors.

We suggest that the time for message-passing in a wide range of applications can be estimated by using these models. The message-passing time in an application is estimated by summing the times of its basic communication components.

These models have proved to be useful for tuning high-performance message-passing scientific applications. For example, we have used them to analyze the performance of an explicit method finite element application for vehicle crash simulation [5]. We have also used them in a research study to evaluate and compare the performance of six broadcast algorithms on the SP2 [15].

In this research we have studied standard broadcast algorithms like One-to-many broadcast (otm), binary-tree broadcast (bt), and recursive-doubling broadcast (rd). In otm, the sender processor sends the message to all the receivers one at a time. In bt, the processors are arranged in a binary tree with the sender process on the root. Each processor receives the message from its parent (if it has one) and sends it to its children (if it has any). In rd, the processors are arranged in a hypercube and do communication in stages. First the sender sends to its neighbor in the first dimension, then the two processors send to their neighbors in the second dimension, and so on doubling the number of processors having the message after each stage.

We have shown that the broadcast time for each algorithm can be approximated by the basic communication models. For example, otm time is \( T_{1m}(n, p - 1) + T_{pp}(n) \), i.e. sender busy time for the first term plus complete time for the last message. Similarly, bt time is \((T_{1m}(n, 1)) + T_{pp}(n)\) and rd time is \((\text{hypercube dimensions}) 	imes T_{pp}(n)\). We have found that no single algorithm is optimum for all ranges of \( n \) and \( p \), but a hybrid broadcast algorithm can contain the best algorithms (otm, rd, and a pipelined bt) and use their models to select the best, given \( n \) and \( p \).

These models can also be used to compare different multicomputers. Developing these models for different multicomputers reveals many aspects of their relative performance. We have conducted one such communication performance comparison [15] by developing the performance models for the PVM message-passing library on the IBM SP2 and the Convex SPP-1000 [16]. This study has shown that the two systems have somewhat similar communication performance, with some notable differences.

Synchronization in a multicomputer occurs implicitly whenever the processor nodes transfer data through message passing. This synchronization results in increased overhead when the load balance among the processors is poor.

References