

# LoGPC: Modeling Network Contention in Message-Passing Programs

Csaba Andras Moritz and Matthew I. Frank

**Abstract**—In many real applications, for example, those with frequent and irregular communication patterns or those using large messages, network contention and contention for message processing resources can be a significant part of the total execution time. This paper presents a new cost model, called LoGPC, that extends the LogP [9] and LogGP [4] models to account for the impact of network contention and network interface DMA behavior on the performance of message passing programs. We validate LoGPC by analyzing three applications implemented with Active Messages [11], [19] on the MIT Alewife multiprocessor. Our analysis shows that network contention accounts for up to 50 percent of the total execution time. In addition, we show that the impact of communication locality on the communication costs is at most a factor of two on Alewife. Finally, we use the model to identify trade-offs between synchronous and asynchronous message passing styles.

**Index Terms**—Multiprocessors, modeling, pipelining, contention, network.

## 1 INTRODUCTION

USERS of parallel machines need good performance models in order to develop efficient message passing applications. Several approaches to modeling the communication performance of a multicomputer have been proposed in the literature [4], [9], [15]. These models often capture some, but not all, of the aspects of the parallel machine. This paper presents a new cost model, LoGPC, that extends the LogP [9] and LogGP [4] models with a model of network contention delay. Our primary objective is to present a methodology of analyzing application behavior when contention effects are present.

We use an engineering approach to performance modeling, as shown in Fig. 1. This approach uses three layers, incrementally incorporating more details about system architectures and application programs. The top layer represents general models that capture first order system costs, like LogP and LogGP do. The second layer extends the first one by including more details about the modeled architecture. For example, memory and network interfacing could be modeled in more detail. These extra details could be used to refine the base performance models by adding new parameters or giving a better estimation for existing ones. These performance metrics capture an ideal state of the parallel system. The communication patterns of many applications, however, cause contention delays. The third layer includes models for estimating contention delays based on application specific parameters.

- C.A. Moritz is with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, Amherst, MA 01002. Email: andras@ecs.umass.edu.
- M.I. Frank is with the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139. Email: mfrank@lcs.mit.edu.

Manuscript received 03 Dec. 1998; revised 09 June 2000 accepted 27 June 2000.

For information on obtaining reprints of this article, please send e-mail to: [tpds@computer.org](mailto:tpds@computer.org), and reference IEEECS Log Number 108371.

The LoGPC model leverages the existing features of the LogP model for fixed-size short messages which have been shown to be successful for regular applications with good communication locality and tight synchronization. In addition, LoGPC uses the features of the LogGP model to account for long message bandwidth. LoGPC extends these models with a simple model of network contention effects. We use Agarwal's open model for  $k$ -ary  $n$ -cubes [1] and close it by including the impact of network contention on the message injection rate. Finally, LoGPC models the pipelining characteristics of DMA engines which allow the overlap of memory and network access times.

We validate LoGPC by comparing its predictions to the measured performance of three applications implemented with Active Messages [11], [19] on the MIT Alewife [2] multiprocessor. The applications used for validation are all-to-all remap with synchronous and asynchronous messaging, a dynamic programming-based DNA chain comparison program called the Diamond DAG, and EM3D, a benchmark code that models the propagation of the electromagnetic waves in solids.

Using our analysis techniques, we were able to identify and then eliminate performance bugs in our original implementation of EM3D which accounted for 20 percent of its runtime. In addition, we show that network contention delays cause the performance of the Diamond DAG application to vary by up to 54 percent depending on how data is mapped to the nodes. Finally, we show that network contention accounts for up to 50 percent of total runtime in the all-to-all remap benchmark.

In addition, we have used the LoGPC model to study two aspects of parallel program design. First, we have studied the impact of locality (i.e., the data and process mapping) and message size on network contention. We show that, given the MIT Alewife machine parameters, the performance effect of communication locality in applications with long messages and uniform message distributions is at most a factor of two.

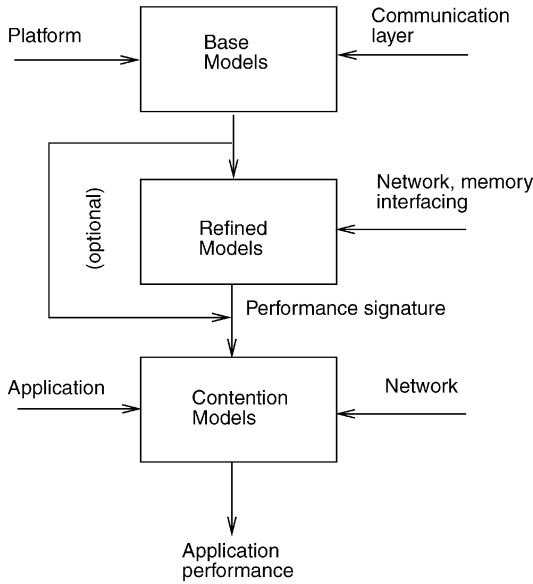


Fig. 1. An engineering perspective to performance modeling of parallel computer systems.

Second, we have examined the trade-off between the contention overheads for synchronous and asynchronous message passing. We find that asynchronous message passing avoids the costs of Network Interface (NI) contention under uniform message destination distributions, but that synchronous message passing is preferred under skewed distributions because it avoids creating network hotspots.

While LoGPC is highly accurate in its runtime predictions (within 12 percent for the communication patterns we studied), we believe that its main value lies in its usefulness for these kinds of engineering trade-off studies. The marriage of simple cost models, like LogP, with simple contention models allows us to study a range of trade-offs between computation, communication, and contention when designing new parallel applications.

### 1.1 Overview

The remainder of this paper is organized as follows: Section 2 discusses the contention-free performance model with the LoGPC extension for pipelined DMA. Section 3 describes the network contention model used in LoGPC. Section 4 discusses how the model may change assuming different message receptions, different NI architecture and network models, and different implementations of messaging. In Section 5, we discuss the experimental results, comparing the LoGPC predictions with measured application performance on the MIT Alewife multiprocessor. Section 6 discusses related work, and Section 7 concludes the paper.

## 2 CONTENTION-FREE COMMUNICATION PERFORMANCE

This section discusses and parameterizes the communication performance on the MIT Alewife multiprocessor using Active Messages in the absence of network contention. In the first part, we discuss the LogP [9] parameterization of

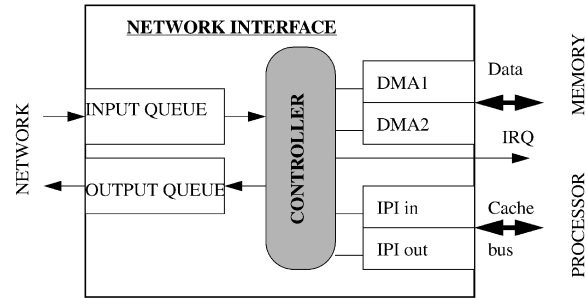


Fig. 2. Alewife network interface. The two DMA engines move user level messages between the network queues and the memory. The controller interrupts the processor whenever a new message arrives in the input queue.

the Alewife machine for short messages. For short messages, and ignoring contention, LoGPC is equivalent to LogP. In the second part of this section, we show how we model the performance of long messages, again, without accounting for contention. For long messages, LoGPC uses the same parameters as LogGP [4] and, in addition, takes DMA pipelining into account. The performance parameters derived here are used in later sections to derive the contention components for different applications.

The MIT Alewife multiprocessor has a no end-around asymmetric mesh network with bidirectional channels using wormhole routing. Each node has an integrated shared-memory and message passing interface with 256-byte network input and output queues. The architecture of the communication controller [2] for message passing (we ignore the shared memory support) in each Alewife node is shown in Fig. 2. The two DMA engines support efficient message transfer between network queues and memory.

The message passing layer used is interrupt-based Active Messages. A message will include the address of the receive handler that is executed after the interrupt is processed. The receive handler has higher priority than the background thread, i.e., a message will interrupt the execution of the running thread on the receiving processor.

### 2.1 Short Message Performance

For short messages, the LoGPC model uses the performance parameters of the LogP [9] model. LogP is a simple parallel machine model that reflects the most significant factors affecting the performance of traditional message passing computers. The LogP model parameterizes parallel machines in terms of four parameters:

1.  $L$  = Latency, or the upper bound on the time to transmit a message from its source to destination in an unloaded network.
2.  $o$  = overhead, or the time period during which the processor is busy sending and receiving a message.
3.  $g$  = gap, or the minimum time interval between consecutive sends and receives.
4.  $P$  = Processors, or the number of processors.

The model also assumes a network with a finite capacity, i.e., if a processor attempts to send a message that would exceed the capacity of the network, the processor stalls

TABLE 1  
LogP Short Message Parameters on the  
MIT Alewife Multiprocessor (in Cycles)

No. of arguments	$L$	$o_s$	$o_r$
null arg message	21	9	117
1 argument	21	12	119
2 argument	21	15	122

until the message can be sent. The model is asynchronous, i.e., processors work asynchronously and the latency experienced by any message is unpredictable, but, in an unloaded network, bounded above by  $L$ .

The network latency,  $L$ , for short messages in the LogP model is defined to be the time after the sending processor is finished performing the send operation and before the receiving processor is interrupted with notification of the arriving message. On Alewife, this parameter corresponds to the time required for the first byte of the message to make its way through the network plus several additional cycles for the entire 8-byte header to arrive at the network interface before the receiving processor is interrupted. Additional latency for message payload beyond the header can be ignored because the arrival overlaps with the processor's interrupt processing (and is, therefore, included in the  $o_r$  parameter).

The  $g$ , or "gap" parameter represents the maximum rate at which a processor can inject messages in the network and captures the node to network bandwidth. On Alewife, the node to network bandwidth is higher than the maximum rate at which short messages can be composed. Thus, the send overhead,  $o_s$ , can be used to approximate the gap. Our empirical results confirm the validity of this approximation.

The short message LogP performance results for the Alewife multiprocessor are shown in Table 1. The send overhead,  $o_s$ , is very small, and the receive overhead,  $o_r$ , is essentially the cost for taking the interrupt. As explained above, the latency,  $L$ , is constant as message length grows because the arrival of any data after the header is overlapped with the receive interrupt handler.

## 2.2 Long Message Performance and Pipelining

For long messages, LogGP uses the same parameters as LogP, and extends the model to account for pipelining in the DMA unit. The LogGP [4] model is an extension of LogP for long messages. It accounts for long message support with an additional parameter,  $G$ , or *Gap per byte*, where  $1/G$  is the network bandwidth in bytes per unit time.

As shown in Fig. 2, each Alewife node has two DMA channels which are used exclusively for long message sending and receiving. The sending DMA is programmed by writing special control words into the memory mapped IPI output registers. This is followed by the processor issuing a *send* instruction to start the message injection into the network.

The network interface interrupts the processor each time a message is received at the network input queue. The processor examines the message header and starts transferring long messages to memory by issuing an instruction to start the receive DMA engine. The send and

TABLE 2  
LogGP Long Message Parameters with Bulk Transfer Active  
Messages on the MIT Alewife Multiprocessor (in Cycles)

$L$	$o_{sl}$	$o_{rl}$	$1/G$	$G$
8	25	129	41MB/s (2bytes/cycle)	0.5 cycles/byte

Both the send overhead,  $o_{sl}$ , and receive overhead,  $o_{rl}$ , include the 10 cycle cost of initializing the DMA engine. The message receive overhead also includes the 119 cycle cost of an interrupt.

receive DMA engines can operate simultaneously, although they will contend for the memory port. When it occurs, this memory contention causes a 45 percent slowdown for each DMA operation.

The send and receive DMA operations may be either blocking or nonblocking. In *blocking* mode, the processor starts the DMA operation and then waits for the entire memory transfer to finish before continuing. This is useful if, for buffer management or synchronization, the processor needs to know when the memory transfer has completed.

If there are other threads available to overlap the memory transfer, the program may instead choose to issue a *nonblocking* DMA operation. In this mode, the processor starts the DMA operation but then continues with other work, permitting the overlap of communication and computation.

The measured LogGP parameters for long messages are shown in Table 2. The processor overheads for large messages are larger than for short messages as they also include DMA setup operations. Both sending and receiving a message includes a 10 cycle cost for initializing the DMA engine. The message receive overhead also includes the 119 cycle cost of an interrupt. Fig. 5 shows the bandwidth for various message sizes with bulk transfer.

Sending a message of  $B$  bytes first involves spending  $o_{sl}$  cycles in launching the message, including the cost for DMA setup. In the nonblocking version, the sender processor is busy only during this time. Each byte travels  $L$  cycles before reaching the destination processor. As with short messages, for long messages in the LogGP model, we define the network latency  $L$  to be the average latency of a message header in an unloaded network. The measured latency for long messages is not the same as that for short messages because, for long messages, we include only the cost of network traversal for the first byte and not the time for the rest of the header packet.

Bytes subsequent to the first take  $G$  cycles to enter the network, as shown in [4]. After the first few bytes of the header arrival (shown as  $a$  in Fig. 3), the network interface generates a processor interrupt, which runs a receive handler. The handler starts up the DMA transfer and then the processor returns to the main thread interrupted by the receive handler. During the time the interrupt is taken, message data continues to arrive and is queued in the network input queue.

The DMA engine on the sending node, the wormhole routed network, and the receiving DMA work in parallel as a three stage pipeline to provide nearly minimal end-to-end delivery times. This pipelining is demonstrated in both Figs. 3 and 4. Because the memory bandwidth on Alewife is nearly twice the network router bandwidth, data is

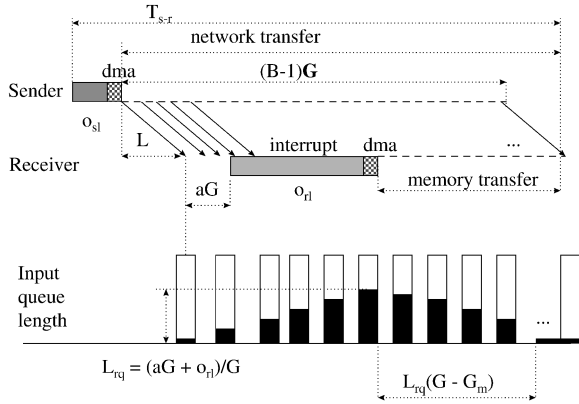


Fig. 3. Input queue length as a function of time during pipelined long message transfer on the MIT Alewife multiprocessor. An interrupt is generated only after the first several bytes arrive (time  $aG$ ). Message data continues to arrive during the time the interrupt is being processed and before the DMA transfer is started (time  $o_{rl}$ ). The memory transfer bandwidth,  $G_m$ , is somewhat better than the peak network bandwidth,  $G$ , so eventually the queue empties and data is transferred to memory at the slower network rate.

transferred to memory faster than it arrives in the input queue. For sufficiently long messages (about 1,000 bytes), any queue buildup that occurs during the initial interrupt will be drawn down by the time the final byte arrives.

For these messages, the network bandwidth is the limiting factor. The total *end-to-end* delivery time, defined to be the difference between the time at which the sending node first starts the send operation to the time the receiver receives the last byte, will be given by

$$o_{sl} + L + (B - 1) * G. \quad (1)$$

Here,  $o_{sl}$  is the time for the sender to initiate the message,  $L$  is the average time for the message header to travel through the network,  $B$  is the message length (in bytes), and  $G$  is the network "Gap" (in cycles per byte). (See Table 3 for a summary of notation.)

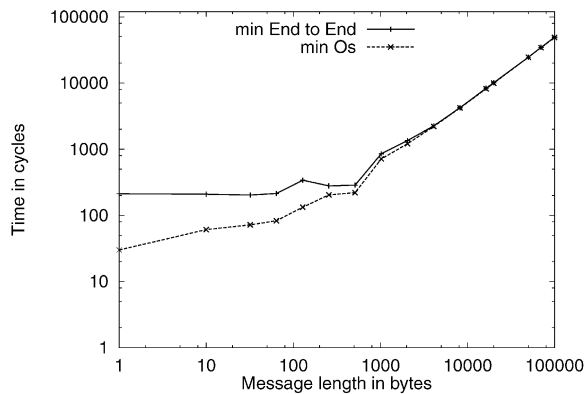


Fig. 4. Network bandwidth swamps interrupt cost. The upper curve ("min End to End") shows total delivery time from initiation on the sender to the delivery of the final payload byte on the receiver including the cost of interrupting the receiver. The lower curve ("min  $O_s$ ") shows the cost of performing only the sender side operation, from the initial send operation to the injection of the final byte into the network. For messages larger than 1,000 bytes, the two times are nearly equivalent because the fast memory transfer drains the queues that filled during the interrupt.

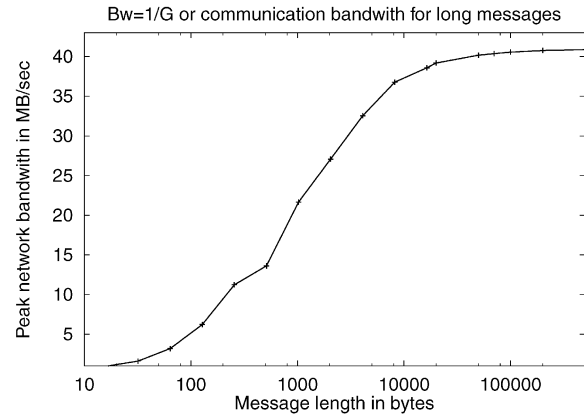


Fig. 5. Network bandwidth,  $1/G$ , for different message sizes. For sufficiently large messages (over about 10 Kbytes), the network bandwidth can be approximated as constant.

For messages smaller than 1,000 bytes, the cost of the initial interrupt is also a factor. For these messages, the end-to-end delivery time is  $o_{sl} + L + aG + o_{rl} + BG_m$ . Here,  $o_{rl}$  is the cost of handling the interrupt and  $G_m$  gives the memory transfer rate to move the queued data to memory.

In general then, the total end-to-end message delivery time from sender to receiver, which we denote with  $T_{s-r}$ , is given by:

$$T_{s-r} = o_{sl} + L + \max(o_{rl} + aG + BG_m, (B - 1)G). \quad (2)$$

For the applications considered in this paper, the message length is always larger than 1,000 bytes and we use the simpler (1).

### 3 NETWORK CONTENTION

This section presents a methodology for extending LoGPC to account for network contention. The basic approach we take is to begin with the LogGP machine parameters along with information about a specific program's messaging rate, and to then apply these parameters to a queuing model to calculate the network contention observed by the program.

TABLE 3  
Summary of Notation

$n$	network dimension
$k_d$	average distance in each direction
$k_{dl}$	average distance in each direction for a locality scheme $l$
$\rho$	probability of a message arriving at an incoming channel
$B$	message size in bytes
$L$	no contention network latency
$L'$	network latency including network contention
$a$	the number of bytes before interrupt is taken
$w_b$	delay in a switch due to contention for $B$ bytes
$C_n$	network contention per message
$C_r$	resource contention per request-reply cycle
$m$	message injection rate not including contention delay
$m_c$	message injection rate including contention delay
$T$	inter message time
$T_c$	inter message time including contention delay
$T_{s-r}$	end-to-end message delivery time
$o_s, o_{sl}$	send overheads for short and long messages
$o_r, o_{rl}$	receive overheads for short and long messages
$G$	gap per byte on the network for long messages
$G_m$	gap per byte for memory transfer for long messages

Our technique uses Agarwal's open model for  $k$ -ary  $n$ -cubes [1] to calculate the network contention from the message injection rate, and then closes the model by feeding the network contention costs back into the calculation for the message injection rate. This section begins by giving a brief overview of Agarwal's model, (3) through (6). Then, we discuss the method for closing the model. Next, we give a number of specifics for dealing with short and long messages and locality. The section closes with a discussion of a method for deriving an upper bound on the effects of network contention.

In this paper, we consider directly buffered  $k$ -ary  $n$ -cube networks that use *wormhole* routing. In wormhole routing, when the header of a message is blocked, all other data in the message stops advancing and remains in the network, blocking the progress of any other message requiring the channels occupied. Our objective is to provide a methodology of predicting network contention in message passing applications.

The average distance (with randomly chosen message destinations) a message travels in each network dimension is denoted by  $k_d$ , thus, the average distance in an  $n$ -dimensional network is  $nk_d$ . The average delay through a switch is derived by Agarwal in [1] from a set of equations that result from the M/G/1 queuing system. We assume a minimal routing algorithm, i.e., the message header is routed completely in one dimension before the next. The average waiting time, as a function of the probability of a message arriving at an incoming channel, and the length of the message size is given by the following expression:

$$w_b = \frac{\rho B}{1 - \rho} \frac{k_d - 1}{k_d^2} \left(1 + \frac{1}{n}\right). \quad (3)$$

This equation could also be expressed as a function of the probability of a network request,  $m$ , in any given cycle on a processor. The probability of a message arriving at an incoming channel,  $\rho$ , is a function of  $m$  and can be determined as follows: The message must travel a distance of  $nk_d$  on the average. With  $B$  bytes, the probability of a byte arriving at an incoming channel also increases by a factor of  $B$ . Because each switch has  $n$  channels, and assuming that we have bidirectional channels implemented as two physical channels, the probability of a message arriving at an incoming channel is:

$$\rho = \frac{Bmnk_d}{2n} = Bmk_d/2. \quad (4)$$

Thus,

$$w_b = \frac{mB^2/2}{1 - mBk_d/2} \frac{k_d - 1}{k_d} \left(1 + \frac{1}{n}\right). \quad (5)$$

The above equation describes the average per node contention delay caused by a message of  $B$  bytes traveling  $k_d$  distance in each dimension and assuming a message injection rate,  $m$ . The total network contention delay per message traveling an average of  $k_d$  distance in each of the  $n$  dimensions is:

$$C_n = nk_d w_b. \quad (6)$$

Assuming we know the no contention message rate, we can compute the average delay due to contention. Although this approximation may be acceptable for short messages, for long messages, we easily obtain a message injection rate that is higher than the saturation rate. In order to be able to compute the delay due to contention, we need to feed back the contention delay into the message rate (as the message rate decreases with contention). This is an extension to the network model presented in [1], where the message injection rate was considered constant. The following equation describes this situation:

$$\frac{1}{T + C_n} = m_c. \quad (7)$$

The message rate without contention is  $\frac{1}{T}$ . Assuming a message of size  $B$  sent on average in one iteration, each with  $C_n$  contention, then  $m_c$  gives the average message rate with contention. The above equation is a quadratic equation in either  $\rho$  or  $m_c$ . Solving this quadratic equation gives the solution for the contention with large messages.

### 3.1 Short Messages

In the LogP model, the cost for sending a short message from source to destination, not including contention, is  $o_s + L + o_r$ . This time is inflated by  $C_n$  when contention delay is considered.

$$T_{s-r} = o_s + L' + o_r = o_s + L + C_n + o_r. \quad (8)$$

Replacing  $C_n$  with its expression in the previous equation, we obtain the transfer time of a short message including the cost for network contention. The message size  $B$  is the size of the short message in bytes.

$$T_{s-r} = o_s + L + \frac{(n+1)(k_d-1)B^2m_c/2}{1 - m_cBk_d/2} + o_r. \quad (9)$$

In applications with high message injection rates, network contention can also cause blocking in the communication controller. This causes inflation of the send overhead for the next message being sent. The inflation of the overhead depends on the message injection rate and the total amount of buffering available at the sender, at the receiver, and in the network.

On the other hand, as the send overhead increases the message injection rate,  $m$  decreases, and so, the likelihood of network contention decreases. As we show in Section 5, the problem of send operations blocking becomes particularly severe with asynchronous message passing if the communication pattern develops hot spots.

### 3.2 Long Messages

The long message transfer time with contention effects can be derived in a similar way. The time at which the entire message of length  $B$  is available at the receiving processor is:

$$\begin{aligned} T_{s-r} &= o_{sl} + (B-1)G' + L' \\ (B-1)G' + L' &= (B-1)G + L + C_n. \end{aligned}$$

Network contention inflates the network latency a message observes as well as reduces the observed network

bandwidth. We denote the inflated L and G parameters with  $L'$  and  $G'$ . On average, we assume an inflation of both  $L$  and  $G$  such that the sum of all inflation during transmission equals the total contention delay  $C_n$ . A better way to account for the contention is to consider only the impact of blocking the message header and keeping the  $G$  parameter constant for the rest of the message. The intuition behind this is that once the header of a message arrives to the destination processor, the rest of the message can proceed without contention because of wormhole routing. The network contention,  $C_n$ , can be obtained by solving (7). The final expression for the transfer time can then be obtained by replacing, in  $C_n$ , the message injection rate,  $m_c$ , obtained previously:

$$T_{s-r} = o_{sl} + (B-1)G + L + \frac{(n+1)(k_d-1)B^2m_c/2}{1-m_cBk_d/2}. \quad (10)$$

### 3.3 Locality

This model can easily be extended to account for communication locality. In applications that exploit locality of communication, the average distance between communicating processors is reduced. Reducing the average distance messages travel improves latency because it reduces the number of hops per message and also reduces network contention. Different applications can use different locality models. We model locality by reducing the average message distance,  $k_d$ . We assume that each message from any processor travels a maximum of  $l$  hops in any direction. Destinations are randomly chosen inside this span.

The Alewife multiprocessor is based on a  $4 \times 8$  no end-around mesh with bidirectional channels. The average network distance (assuming randomly chosen destinations) can be calculated as the sum between the average distances along the  $x$  and  $y$  dimensions. On any dimension  $k$ , the average distance is  $\frac{k^2-1}{3k}$ .

$$2k_d = k_{dx} + k_{dy} = \frac{(k_x + k_y) - \left(\frac{1}{k_x} + \frac{1}{k_y}\right)}{3}. \quad (11)$$

The average message distance on the Alewife mesh is  $2k_d = k_{dx} + k_{dy} = 3.875$ . Using a locality model in which processor  $i$  sends messages to either processor  $i+1$  or  $i-1$  with equal probability will result in an average distance,  $2k_{dl} = 1.875$ . This distance is computed as follows:

$$2k_{dl} = \frac{1}{xy} \sum_{p=0}^{P-1} \frac{\text{hops}(p, p-1) + \text{hops}(p, p+1)}{2}. \quad (12)$$

As the  $x$  dimension is larger than the  $y$  dimension, and  $x$  is the lower dimension, contention effects will be more pronounced on  $x$  compared with  $y$ . As the mesh in Alewife has no end-around connections, contention effects are higher in the central regions compared with margins. The impact of communication locality on the Alewife machine was presented by Johnson in [12].

### 3.4 Upper Bound on Contention

An upper bound on the performance degradation due to network contention can be derived assuming a maximal message injection rate (i.e., the processors do no work, but

simply try to continuously send messages) and by using a mapping with large network distances (e.g., a random mapping). This also gives us an indication on performance improvements obtainable with improved mapping or communication locality for very fine-grained applications with large messages.

If there were no contention, the maximal message injection rate would be  $\frac{1}{2GB}$ . Solving the following equation (obtained from (7)) gives a bound on the message injection rate,  $m$ , with contention:

$$\frac{1}{2GB + \frac{(n+1)(k_d-1)B^2m_c/2}{1-m_cBk_d/2}} = m_c. \quad (13)$$

The solution obtained for  $m_c$  has the form  $m_c = 1/FB$ , where  $F$  is a constant derived from the pair  $k_d, G$ , or  $F = F(k_d, G)$ . The intermessage time with contention,  $T_c$ , is the inverse of the message injection rate with contention, or  $1/m_c$ . A constant upper bound on the inflation of the intermessage time is, then:

$$T_c/T = \frac{FB}{2GB} = \frac{F(k_d, G)}{2G}. \quad (14)$$

Plugging in the Alewife parameters into the above equation gives an upper bound of 2.2 on Alewife.

## 4 DISCUSSION

The network contention model presented in the previous section can easily be adapted to slightly different network topology assumptions. The dimension of the network is captured by the  $n$  parameter. Other aspects of the network, such as end-around connectivity and the type of physical channels used (e.g., unidirectional or bidirectional), can be incorporated by changing  $k_d$ . For example, the average message distance with unidirectional channels and end-around connections in one dimension is  $\frac{k-1}{2}$ . A description of average message distances for  $k$ -ary  $n$ -cubes is presented in [1], for buffered indirect networks in [18].

The communication coprocessor-based communication interface is different only in the reduced processor overhead cost due to a better overlapping between communication and computation. The DMA-based Alewife network interface actually emulates a communication coprocessor as, for relatively long messages, the communication times are completely overlapped with computation. Because message injection rates could potentially be higher in such architectures, the impact of network contention is even more significant.

Applications that use polling driven messaging would need to account for the receive software overheads differently. The analysis for contention effects can be done in a similar way as for the interrupt-based systems.

Network contention effects are less pronounced with high-level messaging systems such as MPI. The explanation is that the communication performance is often limited by the high software overheads.

Parallel computers that implement message passing layers on top of a shared-memory system will still suffer from network contention as the network contention cost is

TABLE 4  
Cost of One Iteration in All-to-All Remap with Two Argument Short Message (in Cycles)

Type	LogP	LoPC	LoGPC	$C_n$	$C_r$	Measured	Blocked CMMU access
Synchronous	316	453	499	23	137	486	0
Asynchronous	137	137	137	31	-	151	0

LoPC [15] is used for estimating processor contention in synchronous messaging. LoGPC includes both costs for network and processor contention. The LogP cost for asynchronous messaging is  $o_s + o_r$ , and for synchronous messaging is  $2(o_s + L + o/r)$ .

mainly determined by the communication volume, which is the same in both cases.

Network bandwidth (and, therefore, contention) on Alewife is fairly representative of what is found in a variety of current generation commercial and research multiprocessors [7]. Because processor speeds are increasing faster than the bandwidth of high-end local area networks (e.g., Myrinet), network contention effects on networks of workstations (e.g., [21]) will be somewhat more severe than the contention observed on Alewife.

## 5 APPLICATIONS

In this section, we validate LoGPC against several applications. The applications studied include an all-to-all remap with either asynchronous or synchronous messaging styles, a dynamic programming-based DNA chain comparison program, called the Diamond DAG, and EM3D, a benchmark code that models the propagation of electromagnetic waves in solids.

Our validations are performed against the MIT Alewife multiprocessor which was parameterized in Section 2.

### 5.1 All-to-All Remap

In this section, we analyze the performance of all-to-all remap with both asynchronous and synchronous short messages, as well as for long messages. By an all-to-all remap, we mean any communication pattern where each processor  $p$  repeatedly sends messages to either a randomly chosen destination, or to destinations  $(i + p) \bmod P$  while increasing  $i$ . The goal of this section is to highlight the key differences between asynchronous and synchronous messaging as well as validating LoGPC for different communication patterns. We focus on the effect of network contention, both on the observed network latency and on the cost of blocking during send operations. For synchronous messages, we also include the delay due to contention for processor resources. We find that asynchronous messaging is much less sensitive to network contention than is synchronous messaging. The network contention model turns out to very accurately predict the performance of the communication.

#### 5.1.1 Synchronous Short Messages

A synchronous short message in this paper denotes a request-reply pair. In some sense, it is similar to a remote read or write primitive in a simple shared memory system. The thread issuing the request waits for the reply message before the next request is issued. A request arriving at a processor is queued if it cannot be serviced immediately (e.g., if another request is already being serviced). In the synchronous all-to-all remap measured here, each processor

repeatedly sends a message to a random destination and then waits for a reply message.

These communication patterns have been previously examined in [15]. That paper introduces an extension to LogP, called LoPC, that uses a multiclass queueing model, based on Mean Value Analysis, to predict contention between different threads for processor resources. For applications using synchronous short messages, overall communication volume is quite low (because each processor can have only one outstanding message at a time), and so contention between messages in the network is not a significant portion of runtime. Rather, LoPC focused on the interference effects of different message handlers trying to run on the same processor at the same time. An empirical result of the LoPC work was that, for communication patterns like synchronous all-to-all remap using short messages, the queueing delay per message sent,  $C_r$ , is approximately equal to the cost of a message handler that sends a reply message ( $o_r + o_s$ ). The cost of each communication roundtrip is then modeled by LoPC as  $2(o_s + L + C_n + o_r) + C_r$ .

We show both the measured times and predictions of several different models in Table 4. For synchronous messaging, the cost of each communication iteration is  $2(o_s + L + C_n + o_r) + C_r$ , where  $C_r$  denotes the average contention for processor resources and  $C_n$  is the additional latency observed due to network congestion. Of the average 486 cycles measured, for each message round-trip, the LoPC model predicts that about 28 percent (137 cycles) are due to contention between message handlers for processor resources. The LoGPC model predicts that an additional 46 cycles ( $2C_n$ ) is due to contention between messages in the network. Added to the base contention-free cost of 316 cycles, given by  $2(o_s + L + o_r)$ , the LoGPC model predicts an average cost per message roundtrip at 499 cycles, a 3 percent overestimate.

#### 5.1.2 Asynchronous Short Messages

The effect of network latency can sometimes be avoided by using an asynchronous messaging model. In an asynchronous all-to-all remap, each processor sends  $n$  messages to  $n$  different locations without waiting for any reply messages. On average, each processor will both send and receive  $n$  messages. We model the total cost of asynchronous all-to-all remap with  $n$  iterations as  $n(o_s + o_r)$ . This communication pattern should be insensitive to network latency because there are no dependencies on the time at which messages arrive.

Table 4 shows the measured and predicted cost per iteration of asynchronous all-to-all remap. The measured cost is about 10 percent larger than predicted by the model. About a third of the difference (5 cycles) is due to

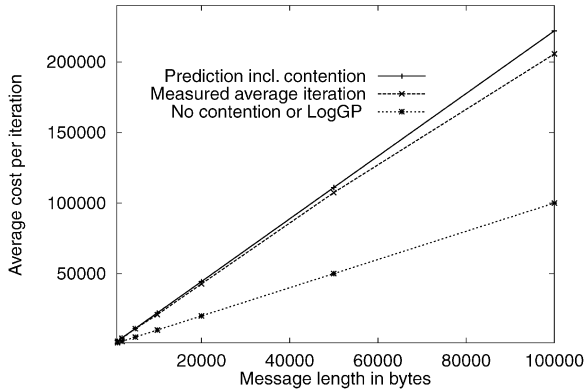


Fig. 6. All-to-all remap for long messages.

unmodeled cache interference effects. The rest of the difference is due to another unmodeled feature of the Alewife communication system. A message that interrupts a send operation incurs extra cost for storing the partially constructed message.

We calculated the average message injection rate for asynchronous messages as  $m = \frac{1}{o_s + o_r}$  assuming that, on average, a send and a receive is served inside each iteration. The last column of Table 4 shows that although messages in this communication pattern incur a considerable amount of extra latency due to network contention, none of the send operations ever blocked because of network flow control. For well-distributed communication patterns like that used here, there are few network hotspots, and Alewife's network output queues are sufficient to hide network contention from asynchronous send operations. Later in this section, we show that the communication pattern from the EM3D application incurs considerable network hotspot contention which does cause send operations to block. Similarly, communication patterns with long messages, as discussed next, can observe considerable cost for blocking sends.

### 5.1.3 Long Messages

Next, we examine an asynchronous all-to-all remap with randomly chosen message destinations and long messages of length  $B$  bytes. Assuming that, on average, one message arrives for each message sent, then the cost of one iteration is  $o_s + (B-1)G + (B-1-a)G$ , or approximately  $2BG$ . See [4] for a similar approximation. As shown in Section 2, on Alewife,  $G = 0.5$  cycles/byte, so the maximum rate is one message injection per  $B$  cycles.

Applying the model in Section 3 (see (13)), we find that in fact the maximum message injection rate, including contention effects, is one message every  $2.22B$  cycles. As shown in Fig. 6, this is a relatively close to the measured message injection rate for all-to-all remap. Across a variety of message sizes, the measured rate shows a small variation around one message per  $2.03B$  cycles.

Note that this analysis has been performed with a maximal message injection rate (i.e., an application that simply tries to send messages as quickly as possible without stopping to do any work). Increasing the computation to communication ratio, or improving the application locality, will decrease network contention.

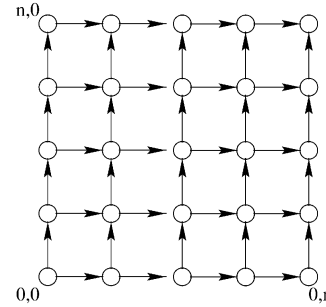


Fig. 7. Diamond DAG.

This permits us to find an upper bound on the effect of contention with randomized communication patterns. Given the Alewife machine parameters, application performance, including network contention, is at most a factor of 2.22 worse than without any contention.

## 5.2 Diamond DAG

In this section, we derive and analyze efficient schedules for the Diamond DAG application. Diamond DAG is a DNA chain comparison program that uses an algorithm based on dynamic programming. This application, Diamond DAG, has been previously analyzed with the *delay* model [22] and the CLAUD model [6].

Our objective in this section is to show how to derive the performance analytically, including the costs of network contention. First, we derive the performance for the no contention case. Then, we calculate the message injection rate including network contention effects. The network contention delay obtained is added to the makespan along the critical path of the DAG. Finally, we compare the analytically derived cost with measurements of an implementation with bulk transfer Active Messages on Alewife.

The Diamond DAG, with  $n \times n$  tasks, can be represented as an  $n \times n$  grid, with vertexes representing tasks and edges representing data dependencies between tasks, see Fig. 7. For the remainder of this section, we will define a unit time as the time to compute a task and express the parameters  $L$ ,  $o$ , and  $G$  in terms of this unit time.

### 5.2.1 Stripe Partitioning

The Gantt Chart in Fig. 8 represents a feasible schedule for the Diamond DAG. Several other partitioning solutions, like line partitioning or a dynamic partitioning of tasks to processors, could also be considered. Finding the optimal partitioning scheme for the Diamond DAG is outside the scope of this paper. Instead we will focus on choosing the correct communication granularity given a stripe partitioning.

In a stripe partitioning, the DAG is partitioned across the processors into  $P$  equal horizontal stripes, each the of size  $(n/P) \times n$ . Each stripe is further split into  $b$  equal rectangular blocks with  $n^2/Pb$  tasks each. Each block depends on, and must wait for, the  $n/b$  results from the block below it in the dependency graph. After all  $n^2/Pb$  tasks in a block have been computed, the  $n/b$  data values along the top edge of the block will need to be sent to the processor that owns the stripe above. Choosing the optimal block size involves a trade-off between the reduced

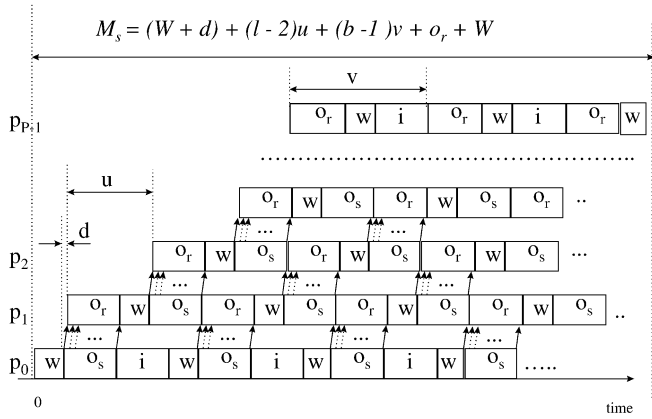


Fig. 8. A Gantt chart of computation and communication schedules for the Diamond DAG with stripe partitioning. The rectangular blocks represent the blocking send ( $o_s$ ) and receive ( $o_r$ ) overheads as well as computation times ( $w$ ). The arrows represent communication between processors.

communication costs obtained by bundling messages and the increased serialization because each processor must wait for the first data set from the previous processor.

We define the *makespan*,  $M$ , as the time consumed for evaluating the entire Diamond DAG. We represent it in the following form

$$M = (W + d) + (P - 2)u + (b - 1)v + O_r + W. \quad (15)$$

This expression follows the critical path in the Gantt chart in Fig. 8. This involves some work on the initial processor,  $W$ , followed by the initial message latency to the second processor,  $d$ . Next,  $u = O_r + W + d$  accounts, on each processor, for the initial cycle of message receiving, work, and latency to the next processor. Finally,  $(b - 1)v + O_r + W$ , where  $v = O_s + W + O_r$  accounts for the processing done in the final stripe.

In the above expression,  $W$  includes both the algorithmic work,  $w_a = \frac{n^2}{Pb}$ , for each block and the extra cost for message aggregation, which we approximate as proportional to the number of data values sent after each block is computed. Thus,  $W = w_a + \alpha \frac{n}{b}$ .

We use  $d$  to denote the period from the time the sender sends the first byte and the receiving processor is notified of the message arrival. This time is  $L + aG$ , where  $L$  is the time needed for the first byte to travel through the network and  $aG$  is the additional time for the complete  $a$  byte message header to arrive at the destination.

Finally, we note that, if the message length per block is  $B = n/b$ , then, using the model from Section 2,  $O_s = o_s + (B - 1)G$  and  $O_r = (B - 1 - a)G$ .

Now, we can find the maximum of  $M$  in  $b$  by solving the equation  $\partial M / \partial b = 0$ .

To account for the costs of network contention, we use the model from Section 3 with the message rate calculated from the equation

$$m_c = \frac{1}{v + 2C_n},$$

where  $2C_n$  is the network contention component inside a block.

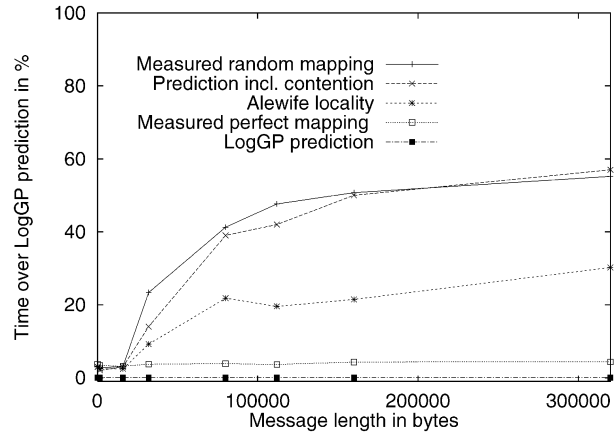


Fig. 9. Diamond DAG performance. Comparison of measured performance to model predictions with stripe partitioning,  $n = 1,024$ ,  $b = 64$ , and each task takes 100 cycles. The “Alewife locality” curve uses the system default node numbering in the Alewife mesh. Node  $i$  sends to node  $i + 1$ , but numerically adjacent nodes are not always adjacent in the mesh.

An upper bound on the cost of contention can then be calculated by assuming that this network contention component is observed by all communication primitives along the critical path. This includes  $P - 1$  sends for the communication in the first block of each stripe and an additional  $b - 1$  sends and receives in the final stripe.

### 5.2.2 Experimental Results

We implemented the Diamond DAG with bulk transfer Active Messages. Although the bulk message mechanism itself is very efficient, the cost for message aggregation and unpacking is very high, accounting for more than 50 percent of total runtime with  $n = 1,024$  and  $b = 4$ . The computational phase of each task consists of two double precision floating point additions and one multiplication.

For the purposes of experimentation, we created a version of the program where the amount of data associated with each block can also be varied. We examined three different mappings of stripes to processors. In the first mapping, stripes were allocated to randomly chosen processors. In the second mapping, stripes were allocated by Alewife’s default node numbering, i.e., stripe  $i$  is mapped to processor  $i$ . While this is better than a random mapping, it is not optimal because the system mapping does not put all sequential processor numbers adjacent to one another in the mesh. In the final mapping, we carefully allocated the stripes such that each processor would communicate only with two physically adjacent processors.

A comparison of measured and predicted values can be seen in Fig. 9. For message lengths up to 16,000 bytes, we observed no contention. The contention-free model provides performance estimates that are accurate to within 3.14 percent. Network contention begins to affect the randomly mapped version when the message size reaches 32,000 bytes per block. This corresponds to a communication to computation ratio of 0.625. As we increased the message length up to 320,000 bytes, the randomly mapped version became up to 56 percent slower than the perfectly mapped version. Using Alewife’s

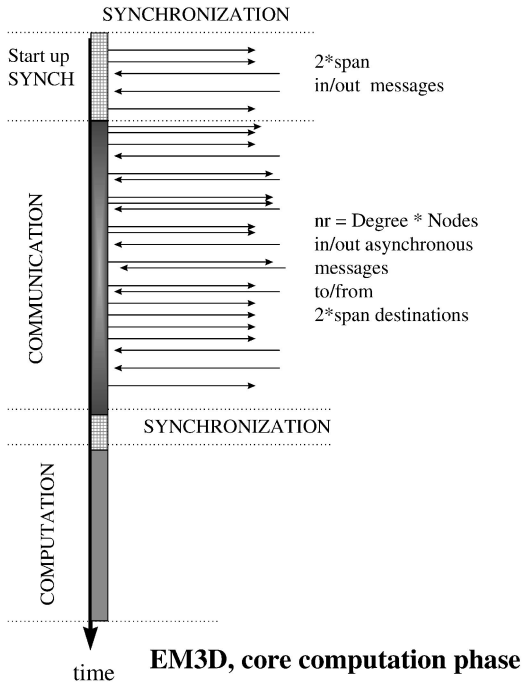


Fig. 10. EM3D core computation phase for E or H nodes.

standard processor, mapping caused up to 31 percent runtime increases due to network contention.

Similar message sizes and communication to computation ratios would also be observed by increasing the parameter of  $n$  (as for very long DNA chains) while keeping  $b$  small. Thus, performance degradation, due to network contention, sets a lower bound on the number of blocks,  $b$ , used for stripe partitioning in the Diamond Dag application.

### 5.3 EM3D

EM3D is a program originally developed at UC Berkeley with the Split-C parallel language. It models the propagation of electromagnetic waves through three-dimensional objects using algorithms described in [20]. We started with a message passing version ported from a CM-5 bulk transfer implementation [8].

EM3D operates on an irregular bipartite graph which consists of E nodes on one side, representing electric values and H nodes on the other, representing magnetic field value at that point. The program has input parameters that can be used to control the total communication volume and the communication locality.

The core phase of the EM3D repeatedly updates the E and H nodes. In each iteration, each node requires the values of all of its neighboring nodes. A processor must send a value for each of its edges that ends on a different processor. In our implementation, values are sent in blocks of 10 using Alewife's short-message facility. This technique is also described as ghost nodes or software caching in [8]. The idea is to communicate node values along edges and buffer them at the receiving processors before the computation phase begins (See Fig. 10).

The communication phase in the core computation is similar to the asynchronous short message all-to-all remap

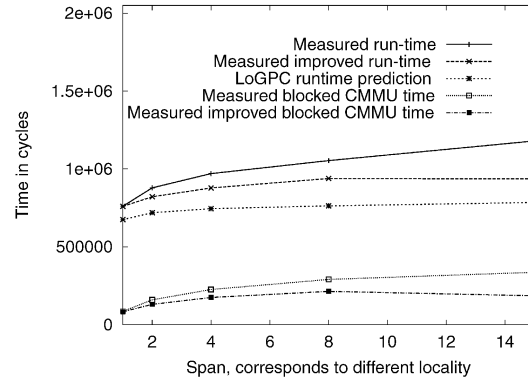


Fig. 11. EM3D performance. Comparison of measured performance to model predictions for the core E + H computation phase of EM3D. On the X axis, the “span” parameter is varied. A larger span corresponds to less locality in the communication pattern. As the communication pattern is asynchronous, when contention hotspots are eliminated, the LogP runtime prediction is the same as the LoGPC prediction. Remaining differences between the LoGPC (or LogP) predicted performance and the measured “improved” performance are due to further hotspots in both the synchronization and communication phases that we were unable to eliminate.

described earlier. We model the communication costs as  $o_s + o_r$  per message.

#### 5.3.1 Experimental Results

Fig. 11 shows a comparison of the measured and predicted runtimes of EM3D as the locality parameter is varied. We used an input data set with 3,200 nodes, degree 32, 100 percent nonlocal edges, 100 iterations and varied the span parameter from 1 to 15. The volume of communication inside the core communication phase is determined by the number of nodes and the degree parameters. The locality of communication is determined by the span parameter. A larger span corresponds to less locality in the communication pattern.

Although the communication phase is similar to the asynchronous all-to-all remap phase described earlier, we observed a much larger effect in the EM3D of network contention, causing send operations to stall. We suspected that the cause of this contention was hotspots in the nonuniform communication pattern. We rewrote the core communication phase so that the message distribution was more uniform (i.e., random). This provided a 70 percent reduction in stall time and a 20 percent overall improvement in runtime. The modified results are labeled “improved” in Fig. 11. The remaining differences between the predicted and measured runtimes are due to further hotspots in both the synchronization and communication phases that we were unable to eliminate.

## 6 RELATED WORK

The LogP model [9] is a simple parallel machine model intended to serve as a basis for developing portable parallel algorithms. Alexandrov et al. defined the LogGP [4] model as an extension of LogP to capture the large bandwidth requirements of applications using long message primitives. LoGPC leverages the performance parameters of LogP and LogGP and extend the analysis with a more

detailed model of the DMA pipeline and a network contention component.

The LoPC model [15] also extended the LogP model with a contention model. LoPC, however, focused on contention between different threads for computation resources rather than on network contention. For the communication patterns studied in that paper (mostly based on synchronous short messages), contention for processor resources accounts for up to a third of total execution time while network contention is not particularly significant. In this paper, we have focused on modeling applications where network contention accounts for a significant portion of total runtime.

The Claud model [6] is similar to the LogP model in the sense that it uses a small set of parameters to model the performance of a message passing computer. Claud attempts to incorporate more details about the interconnection network for a more accurate prediction of network latencies, but does not account for contention effects.

The network contention model used by LoGPC starts with the open queueing model described in [1], but extends and closes the model by relating the terms for message injection rate and message latency. A similar open model of network contention was used by Kruskal and Snir for buffered indirect networks [18]. The impact of communication locality on the Alewife machine using a closed network model similar to ours was presented by Johnson in [12].

A different approach to the question of communication contention can be seen in studies such as [13]. This paper presents a worst case complexity analysis of the nonemptiness problem including the effects of contention. Such studies focus on worst case analysis of particular algorithms where our approach is based on queueing models and attempts to model the constant factors that are of concern to application programmers and machine designers.

A number of researchers have examined application performance in an empirical setting. For example, Karamcheti and Chien [17] studied the network interface architectures in the CrayT3D and TMC CM-5 and examined several messaging implementations for reducing output contention effects. Holt et al. [14] studied the performance of cache-coherent distributed shared memory machines using four parameters similar to LogP. They used the  $\rho$  performance parameter to model the occupancy of the communication controller. Their study shows that application performance is highly sensitive to the controller occupancy. Finally, Chong et al. [7] examined the effect of network bandwidth on application performance using several different communication schemes. They find that message passing communication primitives are less sensitive to network bandwidth than are shared memory primitives.

Several studies apply the contention-free LogP and LogGP models to evaluate the communication performance of various parallel computers and network of workstations. Culler et al. [10] used the LogP model to compare the network interfaces of the Intel Paragon, Meiko CS-2, and a cluster of workstations with Myrinet. Moritz et al. [5] compared the communication performance of MPI on

CrayT3D, Meiko and a network of workstations. Keaton et al. [10] quantified the LogP for local area networks. Martin et al. [21] studied the impact of communication performance of parallel applications in high performance network of workstations. Using the LogGP parameters, they showed that these applications show strong sensitivity to overheads. Finally, Arpaci-Dusseau et al. [3] developed fast parallel sorting algorithms using LogP.

## 7 CONCLUSIONS

Network contention and network interface contention can constitute a large portion of the total run-time of parallel applications. We find that contention accounts for 50 percent of the cost of all-to-all remap with long messages and up to 30 percent of the Diamond DAG, and EM3D benchmarks.

This paper presented a new cost model, LoGPC, that extends the LogP and LogGP models with a simple model of network contention. The network contention model extends and closes the network model described in [1] where message injection rates were considered constant. Although a constant message rate can be used for small messages, it is not applicable for long messages. In addition, LoGPC extends LogGP by modeling network interfaces with DMA support.

For all the communication patterns studied in this paper, the LoGPC model is able to provide performance estimates that are within 12 percent of measured values on the MIT Alewife multiprocessor.

By using the LoGPC model, we were able to study a number of issues in parallel program design. First, we used the LoGPC model to help find performance bugs in our original version of EM3D and as a result we were able to improve the performance of this application by 20 percent. In addition, we have compared asynchronous and synchronous messaging styles. We found that using synchronous message passing caused significant resource contention at the network interfaces. Therefore, asynchronous messaging is advantageous when a uniform message distribution can be guaranteed. On the other hand, nonuniform message distributions can cause serious network contention which causes asynchronous messages, sending primitives to block for long periods of time.

Finally, we studied the impact of application locality on network contention. We find that locality is important up to a constant factor dependent on the network bandwidth and application locality. Using large messages with bulk transfer on the MIT Alewife multiprocessor resulted in a performance degradation up to a factor of two, depending on the mapping used.

## ACKNOWLEDGMENTS

This article is an extended version of the paper that appeared in *Proceedings of the Sigmetrics 1998 Joint International Conference on Measurement and Modeling of Computer Systems*, pp. 254-263, June 1998.

## REFERENCES

- [1] A. Agarwal, "Limits on Interconnection Network Performance," *IEEE Trans. Parallel and Distributed Systems*, vol. 2, no. 4, Oct. 1991.
- [2] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiawicz, B-H. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife-Machine: Architecture and Performance," *Proc. 22nd Ann. Int'l Symp. Computer Architecture*, pp. 2-13, June 1995.
- [3] A. Arpaci-Dusseau, D. Culler, K. Schauer, and R. Martin, "Fast Parallel Sorting under LogP: Experience with the CM-5," *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 8, pp. 791-805, Aug. 1996.
- [4] A. Alexandrov, M. Jonescu, K.E. Schauer, and C. Scheiman, "LogGP: Incorporating Long Messages into the LogP Model," *Proc. SPAA '95*, July 1995.
- [5] C.A. Moritz, K. Al-Tawil, B.F. Rodriguez, "MPI Performance Comparison on MPP and Workstation Clusters," *Proc. 10th Int'l Conf. Parallel and Distributed Computing*, Oct. 1997.
- [6] G. Chochia, C. Boeres, and P. Thanisch, "Analysis of Multi-computer Schedules in Cost and Latency Model of Communication," *Abstract Machine Workshop*, 1996.
- [7] F. Chong, R. Barua, F. Dahlgren, J. Kubiawicz, and A. Agarwal, "The Sensitivity of Communication Mechanisms to Bandwidth and Latency," *Proc. Fourth Int'l Symp. High Performance Computer Architecture*, Feb. 1998.
- [8] D. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick, "Parallel Programming in Split C," *Supercomputing*, Nov. 1993.
- [9] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauer, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," *Proc. Fourth ACM SIGPLAN Symp. Principles and Practices of Parallel Programming*, May 1993.
- [10] D. Culler, L. Liu, R. Martin, and C. Yoshikawa, "LogP Performance Assessment of Fast Network Interfaces," *IEEE Micro*, pp. 35-43, Feb. 1996.
- [11] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauer, "Active Messages: A Mechanism for Integrated Communication and Computation," *Proc. 19th Int'l Symp. Computer Architecture*, pp. 256-266, May 1992.
- [12] K. Johnson, "The Impact of Communication Locality on Large-Scale Multiprocessor Performance," *Proc. 19th Ann. Int'l Symp. Computer Architecture*, pp. 392-402, May 1992.
- [13] A.G. Greenberg, "On the Time Complexity of Broadcast Communication Schemes," *Proc. 14th ACM Symp. Theory of Computing*, pp. 354-364, May 1982.
- [14] C. Holt, M. Heinrich, J.P. Singh, E. Rothberg, and J. Hennesy, "The Effects of Latency, Occupancy and Bandwidth on the Performance of Cache-Coherent Multiprocessors," Technical Report CSL-TR-95, Stanford Univ., Jan. 1995.
- [15] M.I. Frank, A. Agarwal, and M.K. Vernon, "LoPC: Modeling Contention in Parallel Algorithms," *Proc. Sixth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, June 1997.
- [16] K. Keeton, T. Anderson, and D. Patterson, "LogP Quantified: The Case for Low-Overhead Local Area Networks," *Hot Interconnects III: A Symp. High Performance Interconnects*, Aug. 1995.
- [17] V. Karamcheti and A.A. Chien, "A Comparison of Architectural Support for Messaging in the TMC CM-5 and the Cray T3D," *Proc. ISCA '95*, 1995.
- [18] C.P. Kruskal and M. Snir, "The Performance of Multistage Interconnection Networks for Multiprocessors," *IEEE Trans. Computers*, vol. 37, pp. 1091-1098, Dec. 1983.
- [19] K. Mackenzie, J. Kubiawicz, M. Frank, W. Lee, V. Lee, A. Agarwal, and F. Kaashoek, "Exploiting Two-Case Delivery for Fast Protected Messaging," *Proc. Fourth Int'l Symp. High Performance Computer Architecture*, Feb. 1998.
- [20] N.K. Madsen, "Divergence Preserving Discrete Surface Integral Methods for Maxwell's Curl Equations Using Nonorthogonal Unstructured Grids," Technical Report 92.04, RIACS, Feb. 1992.
- [21] R. Martin, A. Vahdat, D. Culler, and T. Anderson, "Effects of Communication Latency, Overhead and Bandwidth in a Cluster Architecture," *ISCA '97*, June 1997.
- [22] C.H. Papadimitriou and M. Yannakis, "Towards an Architecture-Independent Analysis of Parallel Algorithms," *SIAM J. Computers*, vol. 19, pp. 322-328, 1990.



He has consulted and held several industrial positions from chief technology officer to founder.



**Csaba Andras Moritz** received the PhD degree in computer systems from the Royal Institute of Technology, Stockholm, Sweden. He is an associate professor in the Department of Electrical and Computer Engineering at the University of Massachusetts, Amherst. From 1997 to 2000, he was a research scientist at MIT Laboratory for Computer Science. His research interests include modeling, computer architecture, compilers, and scalability issues in distributed systems. He has consulted and held several industrial positions from chief technology officer to founder.

**Matthew I. Frank** received the BS degree in computer science and mathematics from the University of Wisconsin-Madison in 1994, and the SM degree in computer science from MIT in 1997. He is a graduate student in the Laboratory for Computer Science at the Massachusetts Institute of Technology.