Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports
Department of Computer Science

1993

C3: A Parallel Model for Coarse-grained Machines

Susanne E. Hambrusch
Purdue University, seh@cs.purdue.edu

Ashfaq A. Khokhar

Report Number:

93-080

Hambrusch, Susanne E. and Khokhar, Ashfaq A., "C3: A Parallel Model for Coarse-grained Machines" (1993). *Department of Computer Science Technical Reports.* Paper 1093.
https://docs.lib.purdue.edu/cstech/1093

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

C3: A PARALLEL MODEL FOR COARSE-GRAINED MACHINES

Susanne E. Hambrusch
Ashfaq A. Khokhar

CSD-TR-93-080
December 1993
(Revised 2/95)

C^3: A parallel model for coarse-grained machines *

Susanne E. Hambrusch
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA
seh@cs.purdue.edu

Ashfaq A. Khokhar
School of Electrical Engineering and Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA
ashfaq@cs.purdue.edu

February 7, 1995

Abstract

In this paper, we propose a model for parallel computation, the C^3-model. The C^3-model evaluates, for a given parallel algorithm and target architecture, the complexity of *computation*, the pattern of *communication*, and the potential *congestion* arising during communication. A metric for estimating the effect of link and processor congestion on the performance of a communication operation is developed. This metric allows the evaluation of arbitrary communication operations without the user having to specify fine scheduling details. We describe how the C^3-model can serve as a platform for the development of coarse-grained algorithms sensitive to the parameters of a parallel machine. The initial validation of the C^3-model is discussed for the Intel Touchstone Delta. We compare predicted and actual performance of different solutions for communication operations and of various divide-and-conquer approaches for contour ranking on images.

Keywords: Parallel processing, coarse-grained machines, communication operations, computation versus communication, divide-and-conquer.

*Research supported in part by ARPA under contract DABT63-92-C-0022ONR. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing official policies, expressed or implied, of the U.S. government. A preliminary version of this paper appeared in the 6-th *IEEE Symposium on Parallel and Distributed Processing*, October 1994.

1 Introduction

The development of a parallel model that bridges software and hardware has been recognized as crucial to the success of massively parallel computation. Such a model should be simple, should accurately reflect the constraints of a parallel machine, and should have broad applicability with respect to existing machines. In addition, such a model should provide a platform for algorithm development and allow accurate prediction of the preformance of an algorithm. Recently, a number of models with this goal have been proposed [3, 6, 10, 13, 17, 23, 24, 25]. In most of these models, including the BSP model [24], the postal model [3], and the LogP model [6], processors are assumed to communicate using a point-to-point message router. Composing more involved communication operations by using the message router places a significant burden on application programmers. Furthermore, the above models do not attempt to capture the effect of link or processor congestion on communication.

In this paper, we propose a parallel computation model, the C^3-model, for developing and analyzing algorithms on coarse-grained machines. This model captures the complexity of *computation*, the pattern of *communication*, and the potential *congestion* arising during communication. We propose a metric for estimating the effect of link and processor congestion on the performance of communication operations. Parameters of our metric include the number of processors, the number of processor pairs communicating, the latency and the bisection width of the communication network, the message set-up cost, and the packet length. Our metric allows the evaluation of arbitrary communication operations, and it can be applied without having to specify fine scheduling details. We investigate how well the C^3-model serves as a platform for the development of coarse-grained algorithms and as a tool for estimating the performance of an algorithm. We report our initial validation results of the C^3-model on the Intel Touchstone Delta. We compare predicted and actual performance for common communication operations, including one-to-all, all-to-one, and all-to-all routing, and for contour ranking algorithms based on different divide-and-conquer solutions.

In our model, we assume that computation is synchronized by a barrier-style synchronization mechanism similar to the one described in [24]. More precisely, an algorithm can be partitioned into a sequence of supersteps, with each superstep corresponding to local computation followed

by sending and receiving messages. Synchronization occurs between supersteps. We express the performance of a superstep, and thus of an algorithm, in terms of *computation units* and *communication units*. Counting in units allows us to penalize certain undesirable aspects in local computation and in communication. The number of computation units charged depends on the amount of local computation done. The number of communication units charged depends on the amount of data sent by a processor, the amount of data received by a processor, the latency encountered by the messages, and the congestion arising due to the volume of inter-processor communication. Our method for evaluating communication units estimates the effect of these factors on the performance of a communication operation. The routing schemas and routing protocols available on a machine also influence the performance and this is reflected in the total number of communication units charged.

Section 2 describes the C^3-model and the metric devised to determine communication and computation units. In Section 3, we use the model and the metric to determine the communication units for common communication operations when each operations is implemented processors by issuing direct sends and receives. In Section 4, we consider the same communication operations and evaluate and analyze different implementations for each operation. We describe how machine parameters and message sizes influence the performance. In Section 5, we use divide-and-conquer based algorithms for an image-processing problem to validate the C^3-model.

2 The C^3-Model

In this section we describe the metric used by the C^3-model to compute the communication and computation units of a superstep. The parameters of the machine entering the metric are the following:

- p, the number of processors
- h, the latency of the communication network
- b, the bisection width of the communication network
- s, the set-up cost for a message
- l, the length of a packet.

architecture	latency h	bisection width b
linear array	$p/3$	1
binary tree	$\Theta(\log p)$	1
square 2-D mesh	$\frac{2}{3}\sqrt{p}$	$\sqrt{p}$
3-D mesh	$\Theta(p^{1/3})$	$p^{2/3}$
butterfly	$\Theta(\log p)$	$\Theta(p/\log p)$
shuffle-exchange	$\Theta(\log p)$	$\Theta(p/\log p)$
hypercube	$\frac{\log p}{2}$	$p/2$
fat tree	$4\log_4 p$	$\Theta(p)$
complete graph	1	$p^2/4$

Figure 1: Latency and bisection width of machines

We define the latency as the average distance between two processors. The average distance is $(\sum_{0\leq i,j\leq p-1} d_{i,j})/p^2$, where $d_{i,j}$ is the minimum distance between two processors. A message is made up of fixed-length packets and a packet is the logical unit for communication between two processors. The quantity l denotes the number of bytes in a packet. The bisection width is defined as the minimum number of links that have to be removed in order to disconnect the machine into two halves with identical numbers of processors. Figure 1 gives latency and bisection width for various p-processor architectures.

We assume that algorithms on coarse-grained machines are not constrained by the amount of local memory. In current coarse-grained machines, the computing power of a processor is equivalent to that of a state-of-the-art workstation. Hence, for a reasonable problem size, memory is not likely to dictate or heavily influence algorithm design. When describing our metric, we assume that both the processor bandwidth and the network bandwidth are equal to l. How to handle and account for different bandwidth values is described later.

A common feature of parallel algorithms and algorithm design approaches (e.g., divide-and-conquer) is that, at some point or other, the p processors are logically partitioned into q sets $S_1, \ldots S_q$, with S_i containing p_i processors. Communication occurs only between processors in the same set. A programmer familiar with the architecture and the algorithm can often perform a mapping such that communication within processor set S_i does not compete for resources with communication done in the other processor sets. This is possible, for example, when every

processor set S_i corresponds to a scaled down version of size p_i of the p-processor machine. An algorithm then operates on independent submachines. The importance of being able to operate on independent submachines has been recognized. It has been incorporated into the Message Passing Interface (MPI) [8] and has been extended to arbitrary process groups [1]. When it is known that a superstep operates on independent submachines, we charge communication units based on the parameters of the associated submachines.

2.1 Computation Units

The charging of computation units in a superstep is done as follows. Assume that in one superstep processor P_i accesses t_i bytes. At this point we do not distinguish between access to the processor's registers and access to its local memory. However, such distinctions can be incorporated. The superstep is charged $\max_{0 \leq i \leq p-1} \lceil \frac{t_i}{l} \rceil$ computation units. The reason for normalizing computation units by l is that too little computation between two communication steps should have a negative impact on the performance. If $t_i < l$, we charge one computation unit and thus also penalize for not accessing enough bytes to fill a packet.

2.2 Communication Units

The communication units charged to one superstep reflect the time spent in sending messages, the time spent in receiving messages, the time messages are enroute under ideal conditions, the amount of congestion that could occur, and an estimate on the resulting delay. In order to demonstrate broad applicability of our model, we describe the evaluation of communication units for different routing schemas and different send and receive primitives. The two routing schemas we consider are *store-and-forward* and *wormhole routing*. Both are common and they are conceptually quite different. We refer to [14] for details. Most existing machines support both blocking and nonblocking protocols for send and receive primitives. These protocols differ in implementation based on the synchronization methods used. For the sake of completeness, we describe these protocols. A *blocking send* is a send operation initiated by a source processor which does not terminate until the message is received by the destination processor. During this time the source processor cannot perform other computations or communications. In a *nonblocking send* the source processor, after filling its send buffer, has to wait only until the

	nonblocking sends and receives		blocking sends, nonblocking receives	
	send time $s_{i,j}$	receive time $r_{i,j}$	send time $s_{i,j}$	receive time $r_{i,j}$
store-and-forward	$s + \lceil \frac{L_{i,j}}{l} \rceil \star h$	$\lceil \frac{L_{i,j}}{l} \rceil$	$2(s+h) + \lceil \frac{L_{i,j}}{l} \rceil \star h$	$s + h + \lceil \frac{L_{i,j}}{l} \rceil \star h$
wormhole	$s + \lceil \frac{L_{i,j}}{l} \rceil + h$	$\lceil \frac{L_{i,j}}{l} \rceil$	$2(s+h) + \lceil \frac{L_{i,j}}{l} \rceil + h$	$s + 2h + \lceil \frac{L_{i,j}}{l} \rceil$

Figure 2: Send and receive times when P_i sends $L_{i,j}$ bytes to P_j

message has been read out of the send buffer. Nonblocking sends thus allow overlapping of communication and computation and pipelining of multiple send operations. Analogously, receive operations issued by the processors can also be blocking or nonblocking. For additional details on routing protocols we refer to [7, 14, 17].

Sending a single message from P_i to P_j

We start the description of how communication units are determined by giving a cost estimation for sending a single message between two processors. Assume processor P_i sends a message consisting of $L_{i,j}$ bytes (i.e., $\lceil \frac{L_{i,j}}{l} \rceil$ packets) to processor P_j. We charge processor P_i a send time $s_{i,j}$ and processor P_j a receive time $r_{i,j}$. *Send time* $s_{i,j}$ is an estimate on the time needed to send the message when it encounters no congestion. *Receive time* $r_{i,j}$ represents the time processor P_j is occupied with receiving the message. Send and receive times for different routing protocols and routing methods are stated in Figure 2.

The send time includes the time elapsing between issuing the send until processor P_i can resume computation and communication. In addition, it includes the time taken by the message to reach destination P_j. In the case of nonblocking sends, processor P_i could be doing another task at this time. However, a message in transit takes resources away from the machine and the C^3-model charges this to processor P_i. For nonblocking sends and receives with store-and-forward we thus have $s_{i,j} = s + \lceil \frac{L_{i,j}}{l} \rceil \star h$ and $r_{i,j} = \lceil \frac{L_{i,j}}{l} \rceil$. For the case of blocking sends, processor P_i is charged $s + h$ to initiate communication with processor P_j. After that, both

processors are engaged in the sending of the message. Both send and receive time accumulate another $s + h$ when P_j sends a confirmation back to P_i. Processors P_i and P_j are charged the number of units corresponding to the time it takes for the message to reach P_j, resulting in the quantities shown in Figure 2.

Sending multiple messages from P_i

For every processor sending and receiving multiple messages in a superstep, we determine total send and receive times. The total send (resp. total receive) time measures the time a processor is engaged in sending (resp. receiving) messages when messages are not delayed by congestion. We assume that a processor cannot send and receive simultaneously.

Assume that in a superstep processor P_i sends $L_{i,j}$ bytes to processor P_j, $L_{i,j} \geq 0$, $0 \leq i, j \leq p-1$. Let $n_s(i)$ denote the number of processors to which P_i sends a message; i.e., $n_s(i) = |\{j | L_{i,j} > 0\}|$. Let $s_{i,j}$ be as defined above (i.e., it is the cost of sending the message from P_i to P_j without congestion). The total send time, S_i, experienced by processor P_i is an upper bound on the cost for processor P_i to send all $n_s(i)$ messages in a congestion-free environment. Let $r_{j,i}$ be the receive time, as defined above, and let R_i be the total receive time experienced by processor P_i. Further, let $n_r(i)$ denote the number of processors from which P_i receives a message; i.e., $n_r(i) = |\{j | L_{j,i} > 0\}|$. Clearly, $\sum_{0 \leq i \leq p-1} n_s(i) = \sum_{0 \leq i \leq p-1} n_r(i)$. Total send and total receive times depend on the routing schema and the routing protocol used. Figure 3 gives the total send and receive times experienced under different routing protocols.

Consider the case of store-and-forward routing with nonblocking sends and nonblocking receives. Let P_j be the first processor to whom P_i issues a send. After $s + \lceil \frac{L_{i,j}}{l} \rceil$ steps, processor P_i is no longer engaged in the send process for P_j and can proceed with the next send. This allows pipelining the $n_s(i)$ sends. Let $L_{i,j_{max}} = \max_{0 \leq j \leq p-1} L_{i,j}$. The total send time experienced by processor P_i thus contains the $n_s(i)$ start-up costs, the sum of all the packets sent, and $h * \lceil \frac{L_{i,j_{max}}}{l} \rceil$. The final quantity accounts for the latency encountered by the last message to reach its destination. The total receive time is the sum of all the individual receive times. For wormhole routing with nonblocking sends and nonblocking receives, we again pipeline the $n_s(i)$ sends. The latency of the last message shows up as an additive quantity of h.

Protocol*	S_i	R_i
SF, nbs, nbr	$s \star n_s(i) + h \star \lceil \frac{L_{i,jmax}}{l} \rceil + \sum_{0 \le j \le p-1} \lceil \frac{L_{i,j}}{l} \rceil$	$\sum_{0 \le j \le p-1} \lceil \frac{L_{j,i}}{l} \rceil$
WH, nbs, nbr	$s \star n_s(i) + h + \sum_{0 \le j \le p-1} \lceil \frac{L_{i,j}}{l} \rceil$	$\sum_{0 \le j \le p-1} \lceil \frac{L_{j,i}}{l} \rceil$
SF, bs, nbr	$2(s+h) \star n_s(i) + h \star \sum_{0 \le j \le p-1} \lceil \frac{L_{i,j}}{l} \rceil$	$(s+h) \star n_r(i) + h \star \sum_{0 \le j \le p-1} \lceil \frac{L_{j,i}}{l} \rceil$
WH, bs, nbr	$2(s+h) \star n_s(i) + h + \sum_{0 \le j \le p-1} \lceil \frac{L_{i,j}}{l} \rceil$	$(s+h) \star n_r(i) + h + \sum_{0 \le j \le p-1} \lceil \frac{L_{j,i}}{l} \rceil$

Figure 3: Total send and receive times for processor P_i under different routing protocols.
*SF = Store-and-Forward, WH = wormhole routing, nbs = nonblocking sends, nbr = nonblocking receives, bs blocking sends, br = blocking receives

The quantity $S_i + R_i$ represents a bound on the time processor P_i spends in one superstep on sending and receiving messages. Charging one superstep $\max_{0 \le i \le p-1}\{S_i + R_i\}$ communication units reflects the overall send and receive time experienced by the machine during the communication operation, not including the delay the messages encounter because of link and processor congestion. We point out that when stating communication units we have not scaled the set-up cost s, but simply included the total number of set-up costs experienced. When giving communication units for operations on specific machines, as done in Section 4, we convert set-up costs to communication units.

Measuring congestion

We next describe the metric used to estimate the potential congestion arising at the processors or communication links. Congestion plays a crucial role in the time required to complete all routings. At the same time, congestion is difficult to evaluate. Congestion is a global phenomena and where it occurs depends on specifics of the architecture and the routing paths taken. A formal model to deal with congestion in a shared memory machine has recently been proposed in [9]. Congestion depends on the amount of data sent between processor pairs and is

independent of whether we use store-and-forward or wormhole routing. During a routing step, store-and-forward stores K packets in a single processor, while wormhole stores 1 packet (or part of a packet) at K (or more) processors. In our estimation of congestion, we measure C_l, the congestion over links, and C_p, the congestion at the processors. We measure processor and link congestion under the assumption that all messages are routed simultaneously. Clearly, this may not be done under a given protocol. However, delaying the sending of a message by using blocking sends is, in some sense, a possible way of dealing with the congestion. In both cases, the messages experience a delay.

Our metric uses two quantities related to the communication being performed in a superstep. Let *cong* be the total number of processor pairs communicating and let L_a be the average number of packets routed between processors. Congestion over links is closely related to the bisection width of the machine. In a machine with a bisection width of b, it takes at least $\lceil \frac{K}{b} \rceil$ steps to send K packets from processors in one half of the machine to the processors in the other half. We set

$$C_l = L_a \star \lceil \frac{cong}{b} \rceil .$$

Our estimation of the link congestion C_l is both optimistic and pessimistic. It is optimistic in measuring congestion only over a single link cut (namely, the cut that separates the machine into halves). Clearly, link congestion occuring within each half can have an impact on the overall link congestion. It is pessimistic in assuming that all *cong* communicating processor pairs have the source processor on one half and the destination processor is the other half.

In order to estimate the congestion at the processors, assume that all *cong* processor pairs are routed simultaneously. Processor congestion is then estimated as

$$C_p = L_a \star \lceil \frac{cong}{p} \rceil \star h.$$

The quantity $\lceil \frac{cong}{p} \rceil$ represents the average number of messages at a processor at the beginning of the communication operation. We use L_a in estimating the slow-down a message experiences. We argue that a message of size L_a traversing a distance of h links and thus competing for the resources with other messages at each of the $h-1$ intermediate processors is slowed down by a factor of $\lceil \frac{cong}{p} \rceil$ at each processor. We do not take into account that congestion at the processors

is likely to decrease during the routing. Capturing this behavior in a simple way is difficult and in many realistic routings (e.g., a transpose and bit reversal) the decrease in the congestion is slow.

In summary, the total number of communication units charged in a superstep is

$$\max_{0 \leq i \leq p-1} \{S_i + R_i\} + C_l + C_p.$$

In order to estimate actual execution time of an algorithm, relative weights need to be attached to computation and communication units. These weights should be based on the ratio between the processor clock speed and the network clock speed as well as the ratio of the bandwidth of the network and the bandwidth of the processors [18]. In the high-level approach taken by our model, clock speeds and bandwidth parameters do not influence the design of an algorithm and they are thus not included. Put in a different way, we give units for the case when the network clock speed is equal to processor clock speed and network bandwidth is equal to processor bandwidth. When evaluating an algorithm the ratio of computation units and communication units over all supersteps gives information as to whether an algorithm is computation or communication intensive.

3 Charging Examples

Our metric allows evaluation of arbitrary communication patterns. While arbitrary patterns occur in applications, regular patterns are more common on coarse-grained machines. In this section we give the number of communication units charged for regular patterns when each communication operation is implemented using the naive approach of each source processor sending messages directly to the destination processors. The communication operations we consider include one-to-one, one-to-all, all-to-one, and all-to-all routing. The communication units are given for wormhole routing with nonblocking sends and nonblocking receives. To simplify the presentation, we assume that every message is of length L.

One-to-one Routing

In *one-to-one* routing, also known as permutation routing, every processor sends L bytes to a unique destination (i.e., unique among all p processors). Our charging method does not

	S_i	R_i	C_l	C_p
one-to-one	$s + \lceil \frac{L}{l} \rceil + h$	$\lceil \frac{L}{l} \rceil$	$\lceil \frac{L}{l} \rceil \star \lceil \frac{p}{b} \rceil$	$\lceil \frac{L}{l} \rceil \star h$
one-to-all	$(p-1) \star (s + \lceil \frac{L}{l} \rceil) + h,\ i = t$	$\lceil \frac{L}{l} \rceil,\ i \neq t$	$\lceil \frac{L}{l} \rceil \lceil \frac{p-1}{b} \rceil$	$\lceil \frac{L}{l} \rceil \star h$
all-to-one	$s + \lceil \frac{L}{l} \rceil + h,\ i \neq t$	$\lceil \frac{L}{l} \rceil \star (p-1),\ i = t$	$\lceil \frac{L}{l} \rceil \lceil \frac{p-1}{b} \rceil$	$\lceil \frac{L}{l} \rceil \star h$
all-to-all	$(p-1) \star (s + \lceil \frac{L}{l} \rceil) + h$	$\lceil \frac{L}{l} \rceil \star (p-1)$	$\lceil \frac{L}{l} \rceil \lceil \frac{p(p-1)}{b} \rceil$	$\lceil \frac{L}{l} \rceil \star h \star (p-1)$

Figure 4: Communication units charged for wormhole routing with nonblocking sends and nonblocking receives

distinguish between one-to-one routings that are easy or difficult with respect to the arising congestion. Clearly, for any particular architecture, such differences do exist. In one-to-one routing we have $n_s(i) = 1$, $n_r(i) = 1$, $0 \leq i \leq p-1$, and $cong = p$. Figure 4 gives total send and total receive times, link and processor congestion for one-to-one and other communication operations.

For one-to-one routing, link and processor congestion dominate the communication units. Whether one can expect more congestion over the links or at the processors, depends on the bisection width of the machine. Assume that one-to-one routing is done on a p-processor square mesh with $b = \sqrt{p}$ and $h = \frac{2}{3}\sqrt{p}$. Then, processor and link congestion appear almost balanced and we charge

$$s + \frac{2}{3}\sqrt{p} + \lceil \frac{L}{l} \rceil \star (2 + \frac{5}{3}\sqrt{p})$$

communication units. On a p-processor hypercube we have $b = p/2$ and $h = \frac{\log p}{2}$ and the processor congestion dominates. In total, we charge

$$s + \frac{\log p}{2} + \lceil \frac{L}{l} \rceil \star (4 + \frac{\log p}{2})$$

communication units. On a tree machine with $h = \log p$ and $b = 1$ link congestion dominates and we charge

$$s + \log p + \lceil \frac{L}{l} \rceil \star (2 + p + \log p)$$

communication units.

One-to-all Routing

In *one-to-all* routing, a source processor P_t, sends $p-1$ distinct messages, each to a different destination. One-to-all is also refered to as scatter or personalized broadcast [8, 15]. We have $n_s(t) = p-1$, $n_r(i) = 1$ for $i \neq t$, and $cong = p-1$. The total send time experienced by the source processor P_t dominates the number of communication units.

All-to-one Routing

All-to-one routing, also known as the gather operation, is the inverse of one-to-all: every processor now sends a message to a common processor, say processor P_t. We have $n_s(i) = 1$ for $i \neq t$, $0 \leq i \leq p-1$, $n_r(t) = p-1$, and $cong = p-1$. The total receive time at processor P_t dominates the number of communication units.

All-to-all Routing

In *all-to-all routing*, also known as total exchange, every processor sends a message to every other processor. We have $n_s(i) = p-1$, $n_r(i) = p-1$, $0 \leq i \leq p-1$, and $cong = p(p-1)$. From the number of communication units charged shown in Figure 4 it follows that the bisection width of the underlying architecture greatly influences the performance.

4 Validation through Communication Operations

In the previous section we gave the communication units for communication operations when each operation is implemented through source processors issuing direct sends. Such implementations are likely to be used by programmers not familiar with parallel processing. Nor surprisingly, they do not always result in good performance. In this section we use the C^3-model as a platform to develop and analyze different implementations of communication operations. For each implementation we determine computation and communication units and compare total units to the actual performance of the algorithms on the Intel Touchstone Delta. We show that the C^3-model and its metric give an accurate prediction of the relative performance between different implementations of the same operation. Our results also indicate that the performance of

an implementation is influenced by the relationship among parameters of the parallel machine, as well as by the relationship of the parameters to the amount of data involved. This agrees with other research done on the implementation of communication operations [1, 2, 4, 19].

The Intel Touchstone Delta is a coarse-grained multi-processor system with 512 nodes organized as a 16×32 2-dimensional mesh. Each node is directly connected to its 4 nearest neighbors. The communication network uses wormhole routing. Packet size is 512 bytes, with 482 bytes reserved for data and 30 bytes for the message header. The operating system supports both blocking and non-blocking communication primitives. We give communication units and performance for wormhole routing with nonblocking sends and nonblocking receives.

In order to classify different approaches used in our implementations, we introduce the notion of a k-level algorithm. Intuitively, in a k-level algorithm, the machine is partitioned into k levels of submachines, with the submachines within each level operating independently from each other. An algorithm is a *1-level* algorithm if, in the description given in terms of supersteps, no superstep operates on different submachines. In a *k-level* algorithm, $k > 1$, at least one superstep assumes a partition into submachines, not necessarily of identical size, and subsequent supersteps specify a $(k-1)$-level algorithm for each submachine. In our implementations, processors belonging to the same submachine form a scaled down version of the bigger machine. For a mesh, a scaled down version will be either a smaller mesh with the same aspect ratio or a linear array. This is a stronger requirement than the use of process groups as proposed by the MPI Message Passing Standard [8]. When determining communication units, we assume that communication within a submachine occurs without interference from other submachines.

When describing our algorithms, we assume that the size of the message routed between any two processors is L. The objective of our algorithms is to have the processors send out their packets as fast as possible and to minimize the time between processors sending out their last packet and receiving the last packet destined for them. In many situations this time is minimized by combining original messages of size L into larger messages and by performing independent routings in submachines. We refer to L as the *actual message size*. This is in contrast to the *effective message size*, which is the size of the message routed between two processors in a particular superstep. For all algorithms, the effective message size is never

smaller than the actual message size.

4.1 One-to-all Routing

In this section, we use the k-level concept to develop a number of different implementations for one-to-all routing. We evaluate each implementation using the metric of the C^3-model and compare the predicted performance with the performance of the algorithms on the Intel Touchstone Delta.

Description of Algorithms

There exist two conceptually quite different 1-level algorithms for one-to-all routing. In the first one, Algorithm *1-lev-dir*, source processor P_t issues $p-1$ direct sends (and every other processor issues a receive). Using Figure 4, *1-lev-dir* is charged

$$sp + \lceil \frac{L}{l} \rceil \star (p + \lceil \frac{p}{b} \rceil + h)$$

communication units. We point out that throughout this section, we make a number of simplifications when giving communication units. We write p when the correct quantity is $p-1$ and we may omit additive terms of h. Another 1-level approach is to have processor P_t form one long message of size $L(p-1)$ which is broadcast to every processor. After receiving this message, every processor extracts the message destined for it. Our broadcasting implementation, Algorithm *1-lev-br*, uses a binomial heap as a broadcasting tree. One expects the broadcasting approach to be efficient only when L is small and/or when the parallel machine has a control network supporting fast broadcasts. Figure 5 gives an outline of the different algorithms for one-to-all operation.

We next describe a generic 2-level approach. Logically partition the p-processor machine into p^α submachines, each containing $p^{1-\alpha}$ processors for $\frac{1}{\log p} \leq \alpha < 1$. Designate one processor in each submachine as a leader. Source processor P_t then forms p^α long messages, each having an effective message size of $Lp^{1-\alpha}$. The i-th long message formed consists of the $p^{1-\alpha}$ actual messages destined for the processors in the i-th submachine, $0 \leq i < p^\alpha - 1$. Next, processor P_t issues p^α sends (or $p^\alpha - 1$ sends if P_t is a leader) to route the long messages to the leaders. Once

Algorithm 1-lev-dir(p)

The source processor issues p-1 sends, one to each distinct destination.

Algorithm 1-lev-br(p)

1. The source processor concatenates the p-1 messages into one long message which is broadcast. Algorithm *1-lev-our-br* uses a broadcast based on the binomial heap pattern.
2. Each processor extracts its message from the long message received.

Algorithm 2-lev-rec(p)

1. The source processor prepares ($p^{1/2}$-1) long messages, each containing $p^{1/2}$ messages, and sends one long message to each processor in its column.
2. A processor that received a long message, applies Algorithm *1-lev-dir*($p^{1/2}$) within its row.

Algorithm 3-lev-sq(p)

1. The machine is partitioned into $p^{1/2}$ square submachines.
2. The source processor prepares $p^{1/2}$-1 long messages, each containing $p^{1/2}$ messages and sends one long message to each leader processor in the submachine.
3. Each submachine applies Algorithm *2-lev-rec*($p^{1/2}$).

Algorithm logp-lev-sq(p)

1. The machine is partitioned into 2 submachines, alternating partitions along the columns and rows.
2. The source processor concatenates p/2 messages into one long message and sends the long message to the leader processor in the other submachine.
3. Each submachine applies Algorithm *logp-lev-sq*(p/2).

Algorithm logp-lev-rec(p,γ)

1. The machine is partitioned into 2 submachines, one containing γp processors including the source processor, and the other containing (1-γ)p processors.
2. The source processor concatenates (1-γ)p messages into one long message and sends it to the leader processor in the other submachine.
3. The submachine with γp processor applies Algorithm *logp-lev-rec*(γp, γ), and the submachine with (1-γ)p processors applies Algorithm *logp-lev-rec*((1-γ)p, γ).

Figure 5: Outline of one-to-all algorithms.

a leader has received its long message, it divides the message into $p^{1-\alpha}$ of size L and initiates a 1-level one-to-all algorithm within its submachine.

On the Intel Delta we have implemented a 2-level algorithm with $\alpha = 1/2$ in which each submachine is a row containing $\sqrt{p}$ processors. We refer to it as Algorithm *2-lev-rec*. The leaders are the processors in the column containing processor P_t. We use Algorithm *1-lev-dir* as the 1-level algorithm within each row. In Algorithm *2-lev-rec*, the first superstep operates on a single column of the mesh. The second superstep uses Algorithm *1-lev-dir* within each row. The number of communication units charged in both supersteps is

$$s\sqrt{p} + \lceil\frac{L\sqrt{p}}{l}\rceil \star (\sqrt{p} + \lceil\frac{\sqrt{p}}{b'}\rceil + h') + s\sqrt{p} + \lceil\frac{L}{l}\rceil \star (\sqrt{p} + \lceil\frac{\sqrt{p}}{b'}\rceil + h'),$$

where b' and h' are the bisection width and the latency in a $\sqrt{p}$-processor linear array, respectively.

A 3-level algorithm is obtained by applying a 2-level approach to submachines. We considered the following 3-level algorithm, Algorithm *3-lev-sq*, on the Intel Delta. The p-processor machine is logically partitioned into $\sqrt{p}$ submachines, each being an array of size $p^{1/4} \times p^{1/4}$. Once a leader receives its long message from P_l, it initiates a 2-level algorithm for one-to-all routing (using Algorithm *2-lev-rec*) within its submachine.

The value of $k = \log p$ leads to a class of interesting algorithms. A p-processor machine is now divided into two submachines and the source processor P_l issues one send to the leader in the other submachine. If the submachines are of equal size, the effective message size is $Lp/2$. After this send, a $(k-1)$-level algorithm is invoked. If the $(k-1)$-level algorithm proceeds in the same fashion, we refer to the algorithm as a Binomial Heap algorithm (since the sends issued induce a tree having the shape of a binomial heap). When the machine is divided into submachines of equal size, we perform $\log p$ superstep, with each superstep experiencing only a single message set-up cost. Further, the total number of set-up costs experienced is minimized. Algorithm *logp-lev-sq* divides the mesh into half by alternating vertical and horizontal cuts; i.e., the algorithm operates on a square mesh of size $p/4$ after two supersteps. Let $CBH(p)$ be the number of communication units charged to Algorithm *logp-lev-sq* on a p-processor machine. Then,

$$CBH(p) = 2(s+h) + (\lceil \frac{Lp}{2l} \rceil + \lceil \frac{Lp}{4l} \rceil) \star (h+2) + CBH(p/4).$$

For the mesh, the average distance in the $p/4$-processor machine reduces from h to $h/2$. Hence, the recurrence is bounded by

$$CBH(p) \leq s \log p + c \star \lceil \frac{Lp}{l} \rceil \star h,$$

for a constant $c \leq 1.5$.

Algorithm *logp-lev-rec*(γ) divides the mesh into two submachines using γ, $0.5 \leq \gamma < 1$, as the partitioning factor. The partition is made so that the submachine containing source processor P_l consists of γp processors and the other submachine consists of the remaining $(1-\gamma)p$ processors. Evaluating Algorithm *logp-lev-rec*(γ) in the C^3-model results in a larger

Algorithm	Comm. Units	Comp. Units	Comm Units (with s=8)
1-lev-dir	$256s + 0.55L$	$\frac{L}{512}$	$2048 + 0.55L$
1-lev-br	$8s + 27L$	L	$64 + 27L$
2-lev-rec	$32s + 1.23L$	$\frac{L}{32}$	$256 + 1.23L$
3-lev-sq	$24s + 0.93L$	$\frac{L}{25}$	$192 + 0.93L$
logp-lev-sq	$8s + 5.29L$	L	$64 + 5.29L$

Figure 6: Approximate number of units charged for one-to-all algorithms assuming a 256-processor Intel Delta with $h = 10$, $l = 512$, and $b = 16$.

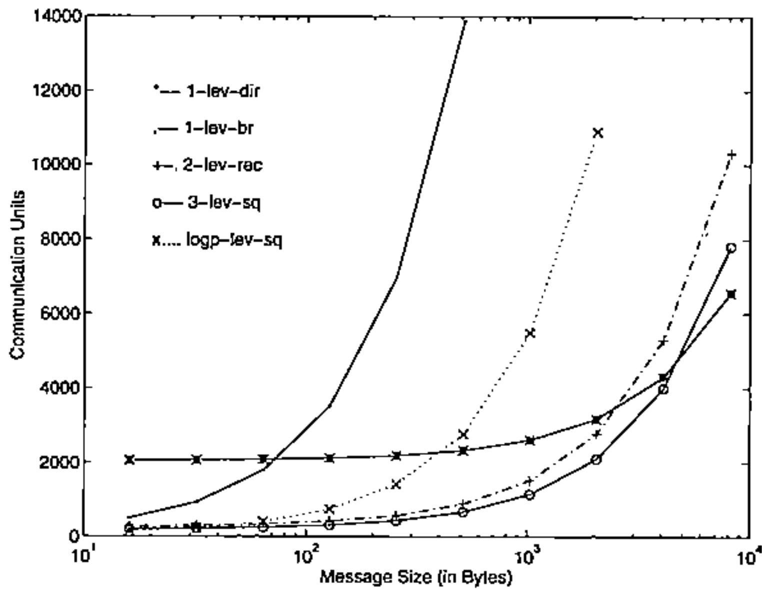
number of communication units compared to *logp-lev-sq*. However, Algorithm *logp-lev-rec*(γ) with $\gamma = 0.75$ performs well on the Intel Delta. We discuss the reasons and why our model fails to evaluates this when comparing actual and predicted performance.

Predicted Performance and Experimental Results

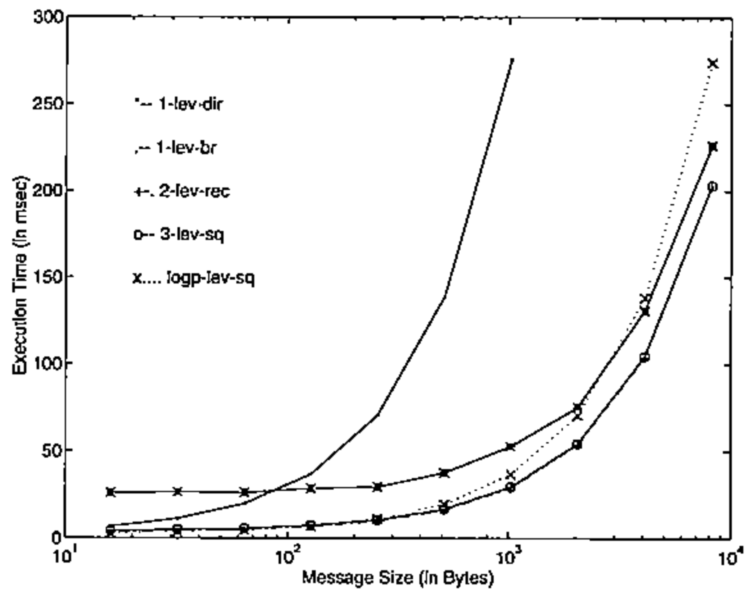
In Figure 6 we show the total number of communication and computation units charged to the one-to-all algorithms in the C^3-model for the Intel Touchstone Delta. The units are given for nonblocking sends and nonblocking receives. Since we considered messages whose sizes are powers of 2, the $\lceil . \rceil$'s have been dropped. The units are given for $p = 256$, $h = 10$ (the precise value would be 10.67), $l = 512$, and $b = 16$. When converting the set-up cost s to units, we assume $s = 1400$ processor cycles. Assuming 40MHz processor clock speed and 12.5 MB/sec network bandwidth, the number of units corresponding to one set-up cost is approximately 8.

Figure 7(a) shows the predicted performance of the algorithms in graphical form, varying the message size from 16 to 16K bytes. From the communication units it appears that Algorithm *3-lev-sq* is the best for message sizes of up to 6Kbytes, and that Algorithm *1-lev-dir* is likely to give reasonable performance for large messages sizes. Algorithm *1-lev-br* is predicted to be a poor choice.

The one-to-all algorithms described in Figure 5 have been implemented on the Intel Touch-



(a)



(b)

Figure 7: (a) Predicted performance (in units) and (b) experimental results (in msec) of the One-to-All Algorithms on a 256-Processor Intel Touchstone Delta using blocking sends and nonblocking receives.

One-to-All Algorithms	Message Size (in Bytes)										
	16384	8192	4096	2048	1024	512	256	128	64	32	16
1-lev-dir	420.82	226.21	130.80	75.22	52.80	37.61	29.50	28.58	26.05	26.45	26.04
1-lev-br	4377.44	2193.51	1096.23	549.12	275.79	138.88	70.67	36.73	19.90	11.29	6.78
2-lev-rec	400.03	203.38	103.79	53.61	28.90	16.25	9.79	6.91	5.06	4.45	3.77
3-lev-sq	402.60	203.30	104.60	54.25	29.46	16.87	10.57	7.56	5.52	4.99	4.17
logp-lev-sq	545.37	274.02	138.30	70.63	36.62	19.62	11.21	6.79	4.46	3.43	2.80
logp-lev-rec(0.75)	393.44	198.13	100.35	51.35	27.14	15.13	9.10	5.80	4.35	3.29	3.02

Figure 8: Performance results for one-to-all routing on a 256-Processor Intel Touchstone Delta using nonblocking sends and nonblocking receives (execution times are in msec).

stone Delta. We considered machine sizes from 16 to 256 processors and message sizes from 16 bytes to 16 Kbytes. The corresponding experimental results for $p = 256$ are shown in Figure 7(b). For a more complete discussion on the performance of these algorithms on the Intel Delta, we refer to [11]. Figure 7 shows that expressing each algorithm in terms of communication and computation units gives an accurate prediction of their relative performance on the Intel Delta. Algorithm *1-lev-dir* is indeed a reasonable choice for large message sizes (at least 4 Kbytes). Independent of the message size, *1-lev-dir* always experiences a total of $p - 1$ message set-up costs. In addition, since the packet length on the Intel Delta is 512 bytes, sending message sizes ≤ 512 costs approximately the same. The broadcasting algorithm gives the worst performance. The poor performance is partly due to the large effective message size, as well as due to the absence of a dedicated fast broadcasting network. Algorithms *2-lev-rec* and *3-lev-sq* give approximately the same performance and are the best choice among the five algorithms listed in Figure 7. Algorithm logp-lev-sq gives good performance only for small message sizes (≤ 256 bytes). This also agrees with its predicted performance. Figure 8 gives detailed performance results in tabulated form.

From Figure 8 is follows that Algorithm *logp-lev-rec*(0.75) performs quite well. Actually, *logp-lev-rec*(0.75) gives optimal or near optimal results for all machine and message sizes on Delta [11]. As already stated earlier, the metric of the C^3-model evaluates *logp-lev-rec*(0.75) to

be no better than Algorithm logp-lev-sq. If Algorithm *logp-lev-rec*(0.75) were implemented with a barrier-style synchronization between supersteps, we would see no improvement. However, *logp-lev-rec*(0.75) was implemented with no such synchronization. The value $\gamma = 0.75$ captures characteristics of the send and receive ratio of the Delta (the value of $\gamma = 0.75$ was obtained through experiments). Before the leader in the other submachine received its long message, the source processor already starts sending the next long message to the next leader. While exploiting such features of a machine can bring good performance results, they are difficult to incorporate into a computational model aimed at making parallel machines easier to use.

In summary, our validation work on the Intel Delta indicates that the message-combining algorithms which keep a balance between the total number of sends and the effective message size perform well for small message sizes. Which one of them gives the best performance depends on the ratio between the send and receive time, the packet length, the ratio between the processor and network bandwidth, and the message set-up cost.

4.2 All-to-one Routing

In all-to-one routing every processor sends a message to destination processor P_t. Processor P_t is now the bottleneck. Conceptually, all-to-one is the inverse of one-to-all. Our one-to-all algorithms, except the algorithm based on broadcasting, have corresponding all-to-one algorithms. Algorithm *1-lev-dir* for all-to-one is one in which every processor issues a send to processor P_t. Algorithms *2-lev-rec* and *3-lev-sq* are the corresponding 2-level and 3-level algorithms, respectively. Algorithm *logp-lev-sq* is the $\log p$-level algorithm partitioning the mesh into two submachines by alternating horizontal and vertical cuts. Algorithm *logp-lev-rec*(γ) partitions the mesh into two submachines based on the value of γ, $0 < \gamma < 1$.

The number of communication units charged for each of the all-to-one algorithms is almost identical to the ones charged for one-to-all and we omit details. The difference lies in the number of message set-ups charged. For example, the communication units charged to Algorithm *1-lev-dir* for all-to-one include only a single message set-up, compared to $p-1$ for one-to-all. For all all-to-one algorithms, the receive times are the dominating terms in the communication units.

From a practical point of view, the best one-to-all algorithms do not necessarily correspond to the best all-to-one algorithms. We refer to [11] for a complete discussion and only state

our main observations. On a 256-processor Intel Delta, Algorithm *1-lev-dir* is no longer a reasonable choice for large message sizes. For a 256-processor machine, all algorithms that combine messages give a comparable performance for $L \leq 512$, while for $L > 512$ Algorithm *logp-lev-rec*(0.60) gives the best performance. For messages of length < 512 bytes the all-to-one algorithms are slightly faster than their one-to-all counterparts, while for messages of length ≥ 512 bytes the all-to-one algorithms are significantly slower. This can be explained by machine characteristics which we do not attempt to capture in the C^3-model.

4.3 All-to-all Routing

In this section, we first describe a number of different algorithms for all-to-all routing. We then compare their predicted performance with the experimental results achieved on a 256-processor Intel Delta.

Description of Algorithms

The most straightforward 1-level approach for all-to-all routing is to have each processor send its $p-1$ messages, one by one, regardless of what other processors are doing. The machine is thus flooded with messages and the arising congestion is left to be handled by the system. This approach is used in Algorithm *1-lev-dir*. An approach that attempts to control congestion implements all-to-all through $p-1$ one-to-one routings; i.e., the $p(p-1)$ routing requests are partitioned into permutations. Common are the linear permutations and exclusive-or permutations. When partitioning into *linear* permutations, processor j sends a message to processor $(j+i) \bmod (p-1)$ in the i-th permutation, $1 \leq i \leq p-1$. When partitioning into *exclusive-or* permutations, all-to-all is partitioned so that in the i-th permutation processor j sends a message to $i \oplus j$. Implementations of these approaches on different machines have shown exclusive-or permutations to be superior to linear permutations [19, 22]. Another interesting approach for partitioning all-to-all routings into permutations has been introduced in [21]. We call this approach partitioning into *balanced* permutations and refer to [11] for implementation details. Balanced permutations are relevant to the mesh architecture since they minimize the congestion over the links.

We view algorithms that partition into permutations as 1-level algorithms and refer to such

Algorithm	Communication Units	Computation Units
1-lev-dir	$256s + 14L$	$\frac{L}{2}$
1-lev-perm	$256s + 14L$	$\frac{L}{2}$
2-lev-sq	$33s + 23L$	$1.5L$
2-lev-c,r	$32s + 23L$	L
logp-lev-bfly	$4s + 28.5L$	$4L$

Figure 9: Approximate number of units charged for all-to-all algorithms on a 256-processor Intel Touchstone Delta with $h = 10$, $l = 512$, and $b = 16$, assuming nonblocking sends and receives.

algorithms as Algorithm *1-lev-perm*. The metric of the C^3-model charges the same number of communication units for each of the three permutations. This is because our metric is unable to distinguish between easy and hard permutations without explicitely giving a partitioning into submachines. Further, our metric charges the same number of communication units for algorithms which partition into p permutations and Algorithm *1-lev-dir*, in which every processor issues $p - 1$ sends independent of what the other processors are doing. The number of supersteps and the amount of congestion in each superstep for both of these 1-level approaches is different, but the total number of units charged is the same. Figure 9 gives the total number of communication and computation units for the all-to-all algorithms.

Next consider the following two 2-level algorithms. The approach used in the first one, Algorithm *2-lev-sq*, is independent of the underlying architecture. The approach used in the second one, Algorithm *2-lev-r,c* is tailored towards the mesh architecture. An idea similar to the one used in Algorithm *2-lev-sq* is described in [4] and an implementation of Algorithm *2-lev-c,r* has also been reported in [22].

In Algorithm *2-lev-sq*, a p-processor machine is logically partitioned into $\sqrt{p}$ submachines, $S_0, \ldots S_{\sqrt{p}-1}$. Submachine S_i performs an all-to-all routing within S_i sending long messages of length $\sqrt{p-1}L$. After this step, processor i in submachine S_j contains the p messages destined for the processors in submachine S_i (and which have their source processor in submachine S_j). The algorithm then performs a one-to-one routing step in which processor i of submachine S_j sends this long message (having length Lp) to processor j in submachine S_i. The third and final step is an all-to-all routing within each submachine which routes the messages to their correct

destinations. Algorithm *2-lev-c,r* uses a similar principle, but avoids a one-to-one routing step by using different submachines in the first and second step. In the first step the $\sqrt{p}$ submachines correspond to the $\sqrt{p}$ columns of the mesh. We perform an all-to-all routing within each column so that processor i in column j receives the p messages destined for the processors in row i (and which have their source processor in column i). An all-to-all routing within each row completes the operation. As shown in Figure 9, the number of communication units charged to the two 2-level algorithms is identical. In Algorithm *2-lev-sq*, $14L$ of the $23L$ units charged come from the second step, the one-to-one routing. In Algorithm *2-lev-c,r* the number of communication units charged is split evenly between the two supersteps.

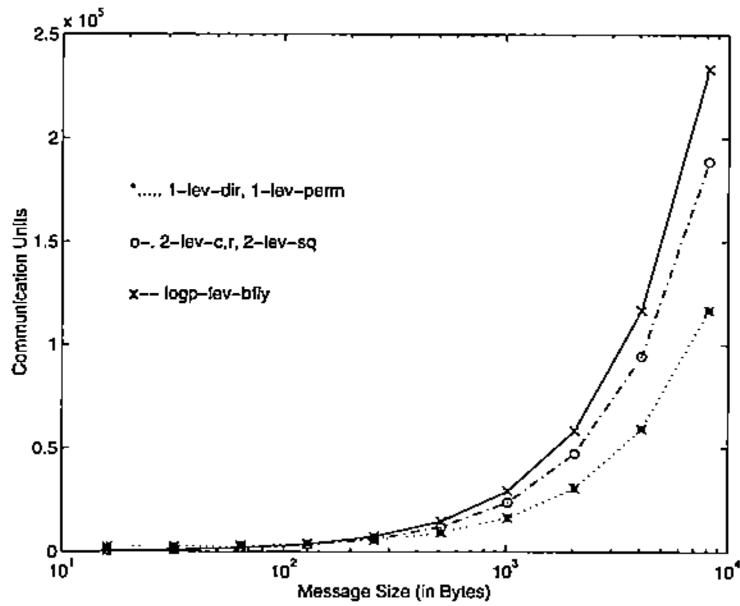
We have also considered a $\log p$-level algorithm, Algorithm *logp-lev-bfly*, based on the butterfly communication pattern. In the first superstep of this algorithm every processor P_i sends the $p/2$ messages destined for the $p/2$ processors not in its half to processor $P_{(i+p/2) \bmod p}$. After the received messages are combined with the messages that remained in a processor, all-to-all in performed on two $p/2$-processor submachines.

Comparing Predicted and Experimental Results

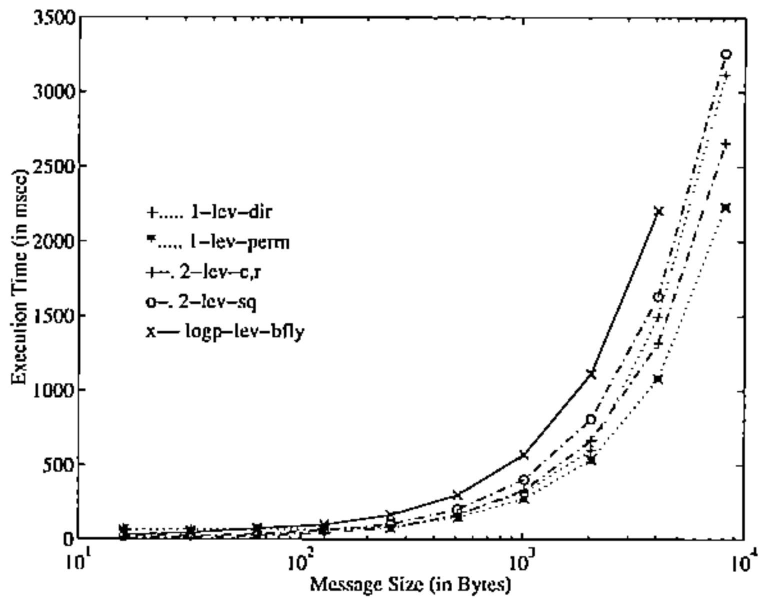
In this section we again compare the performance predicted by the C^3-model to the performance achieved on the Intel Delta. Recall that Figure 9 gives the communication and computation units for the algorithms described in the previous section. The C^3-model predicts the 1-level algorithms to be superior for large message sizes and it predicts message combining algorithms to perform better for small message sizes.

We have implemented the above mentioned algorithms on a 256-processor Intel Delta. Algorithms *1-lev-lin*, *1-lev-Xor*, and *1-lev-bal* are the three 1-level algorithms partitioning all-to-all communciation into permutations. The predicted performace and implementation results are shown in Figure 10(a) and Figure 10(b), respectively.

Algorithm *1-lev-Xor* gives the best performance for large message sizes. Observe that the advantages of Algorithm *1-lev-bal* with respect to the arising congestion are not evident from the experimental results obtained from the Delta. As already stated, the metric proposed in this paper does not distinguish between different 1-level algorithms and thus the predicted per-



(a)



(b)

Figure 10: (a) Predicted performance (in units) and (b) experimental results of the all-to-all algorithms on a 256-processor Intel Delta using nonblocking sends and nonblocking receives.

formance for all 1-level algorithms follow the same curve. However, in actual implementations different permutations induce different patterns of link and processor congestion and thus give a different performance. Capturing this behavior in the model would be difficult.

All-to-All Algorithms	Message Size (in Bytes)										
	16384	8192	4096	2048	1024	512	256	128	64	32	16
1-lev-direct	6860.21	3115.27	1494.48	598.82	316.78	169.48	82.84	73.21	70.28	68.11	69.75
1-lev-lin	5476.28	2661.56	1294.39	639.83	330.90	182.18	94.48	71.12	67.66	63.03	66.55
1-lev-xor	4608.85	2231.60	1081.05	536.01	273.28	147.98	78.20	63.75	59.51	59.21	61.40
1-lev-balance	4988.76	2492.90	1221.90	619.62	305.24	144.25	77.43	77.83	72.83	64.47	61.11
2-lev-sq	6561.45	3260.92	1633.19	809.42	401.09	201.35	99.75	60.03	34.43	24.18	18.69
2-lev-c,r	5632.29	2659.75	1319.53	665.28	330.50	163.02	78.58	39.49	23.46	14.48	11.74
2-lev-c,r-int	4613.42	2232.63	1086.08	543.55	284.23	168.85	113.30	91.23	82.76	78.81	75.96
logp-lev-bfly	--	--	2206.67	1112.07	569.08	298.10	163.34	97.10	74.03	43.09	31.84

Figure 11: Performance Results for all-to-all routings on a 256-processor Intel Touchstone Delta using nonblocking sends and nonblocking receives (execution times are in msec).

The experimental results show that Algorithm *2-lev-c,r* performs best for small message sizes ($\leq$ 256 bytes). Since in Figure 10 it is not easy to distinguish between the performance of the algorithms for small message sizes, we refer to Figure 11. Algorithm *2-lev-sq* gave the second best performance for small message sizes. The reason *2-lev-c,r* outperformed *2-lev-sq*, lies in the fact that *2-lev-sq* is a 3-step algorithm (which sends out data three times), while *2-lev-c,r* is a 2-step algorithm. The advantage of the 3-step algorithm is that it uses square meshes as submachines, whereas the 2-step one uses linear arrays. The approach in Algorithm *logp-lev-bfly* has consistently been judged as being expensive for large message sizes [4, 22]. Our metric and the observed performance on the Delta, confirms that as well.

5 Validation through Divide-and-Conquer Solutions

On coarse-grained machines, divide-and-conquer strategies are natural and often result in efficient solutions. Divide-and-conquer typically contains a merging process in which results

computed by different processors are combined to obtain the final solution. Different merging patterns have different communication and computation requirements. Depending on machine and problem parameters, different patterns are likely to result in different performance.

In this section we use contour ranking, a low-level aimage-processing problem, to validate the C^3-model. Contour ranking can be viewed as performing list ranking in images. The problem arises when edge contours generated by edge operators in a 2-dimensional image plane are transformed into a linearized representation. Such representations are more compact for processing performed in subsequent mid- and high-level vision tasks [5, 16, 20]. Generating the linear representation is called *contour ranking*.

The algorithms we describe use divide-and-conquer and merge information about subimages in order to compute the final values. The information needed about a subimage is proportional to the number of edge points on the boundary of the subimage. The time needed to merge subimages is linear in the number of edge points on the involved boundaries. A number of other problems on images can be solved by algorithms following the same principle. These problems include component labeling, straight line approximations, and region growing. For example, each one of our contour ranking algorithms can be turned into a component labeling algorithm by using a different merging procedure. The relative performance of the so obtained component labeling algorithms will correspond to the relative performance of contour ranking algorithms.

5.1 Problem Definition and Basic Approach

We refer to a pixel on an edge contour as an *edge point*. For each edge point e, $succ(e)$ points to either one of e's eight immediate successors on the edge contour or it is nil. An edge point e with $succ(e) = nil$ is called a *head*. The *succ*-relation induces linked lists and thus each edge contour corresponds to a linked list. In contour ranking we determine, for every edge point e, the head of the list containing e and the distance from e to this head, called the *rank* of e. Once the ranks are known, a final data movement step generates the linear representation. Clearly, by following the *succ*-links, heads and ranks can be determined sequentially in linear time.

Let I be an image of size $m \times n$. For simplicity, we assume that p is a perfect square and that m and n are both multiples of $\sqrt{p}$. We assume that image I is partitioned into p rectangular

subimages, each of size $\frac{m}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$. We number these subimages using a row-major numbering scheme. For clarity, we assume that processor $P_{i,j}$ is assigned subimage $I_{i,j}$, $0 \leq i,j \leq \sqrt{p}-1$.

For any subimage I' of I, the information needed about image I' in order to compute the head and rank information of all edge points outside I' is proportional the number of edge points on the boundary of I'. Conversely, if the final head and rank are known for every edge point on the boundary of I', then the head and rank in image I can be computed for every edge point within I'. In a forward phase, our algorithms merge information about the boundaries of subimages in order to compute the boundary information of larger subimages. In a backward phase, the final head and rank in image I of edge points on the boundary of subimages are used to determine head and rank information for the remaining edge points within the subimages. We refer to [12] for details on how the boundary is represented and for details of the merging. In brief, each one of our algorithms consists of the following three steps.

1. Processor $P_{i,j}$ performs contour ranking on subimage $I_{i,j}$. $P_{i,j}$ then constructs the boundary list representing the information about subimage $I_{i,j}$ needed in future computations.

2. Determine, for each edge point on the boundary of subimage $I_{i,j}$, its rank and head in image I. In order to compute this information, boundaries of subimages are merged.

3. Determine the rank and head in I for every edge point in subimage $I_{i,j}$.

Steps 1 and 3 are identical for each contour ranking algorithm and can be viewed as preprocessing and postprocessing, respectively. They require no communication between processors. In the next section we describe different divide-and-conquer patterns for performing Step 2.

5.2 Divide-and-Conquer Patterns

In this section we describe four algorithms for performing Step 2; i.e., for determining, for each edge point on the boundary of subimage $I_{i,j}$, $0 \leq i,j \leq \sqrt{p}-1$, its head and rank in image I. Assume processor $P_{i,j}$ contains the boundary list of some rectangular subimage $I'_{i,j}$ and processor $P_{k,l}$ contains the boundary list of an adjacent subimage $I'_{k,l}$. Let $I' = I'_{i,j} \cup I'_{k,l}$. In order to determine the boundary list of subimage I', both $P_{i,j}$ and $P_{k,l}$ send their boundary list to each other. After each processor has received the other processor's list, it proceeds to

determine the boundary list of subimage I'. Both processors continue to merge subimages until each processor knows the boundary list of image I. At this point the forward phase of the contour ranking algorithm is completed and the backward phase begins. The goal of the backward phase is to determine, for every edge point on the boundary of subimage $I_{i,j}$, its rank and head in image I. For the algorithms we analyze in this section, the backward phase requires no communication between processors. Processor $P_{i,j}$ uses information about larger subimages to update the heads and ranks of smaller subimages, proceeding until the smaller subimage equals $I_{i,j}$.

Algorithm 1-lev-dir

1. Every processor sends its boundary list to every other processor.
2. Every processor $P_{i,j}$ merges the p boundaries and determines, for each edge point on the boundary of $I_{i,j}$, its the rank and head information in image I.

Algorithm 2-lev-rc

1. Processor $P_{i,j}$ sends its boundary list to every other processor in row i.
2. Processor $P_{i,j}$ merges the received data, creating creating the boundary list of subimage $I_{i,*}$.
3. Processor $P_{i,j}$ sends the boundary list of subimage $I_{i,*}$ to every processor in column i.
4. Processor $P_{i,j}$ determines the boundary list of image I. It then determines the rank and head in I of every edge point on the boundary of $I_{i,j}$.

Algorithm logp-lev-quad

1. Form $p/4$ groups, each containing 4 processors, so that processors $P_{2i,2j}$ $P_{2i+1,2j}$, $P_{2i,2j+1}$, and $P_{2i+1,2j+1}$, $0 \leq i,j \leq \sqrt{p}/2 - 1$ belong to the same group. Number the processors in a group from 1 to 4. Every processor sends its boundary lists to every other processor in the same group.
2. Let $I'_{2i,2j} = I_{2i,2j} \cup I_{2i+1,2j} \cup I_{2i,2j+1} \cup I_{2i+1,2j+1}$. A processor in the same group with $P_{2i,2j}$ determines the boundary lists of subimage $I'_{2i,2j}$.
3. All the processors with number l, $1 \leq l \leq 4$, recursively merge their subimages. After the recursion, every processor in the group with $P_{2i,2j}$ knows the head and rank in image I for each edge point on the boundary of subimage $I'_{2i,2j}$.
4. Processor $P_{i,j}$ determines the rank and head in I of every edge point on the boundary of $I_{i,j}$.

Figure 12: Outline of contour ranking algorithms.

The communication in the forward phase is an all-to-all broadcast performed on submachines. The sizes and types of the submachines depend on the algorithm. We again characterize

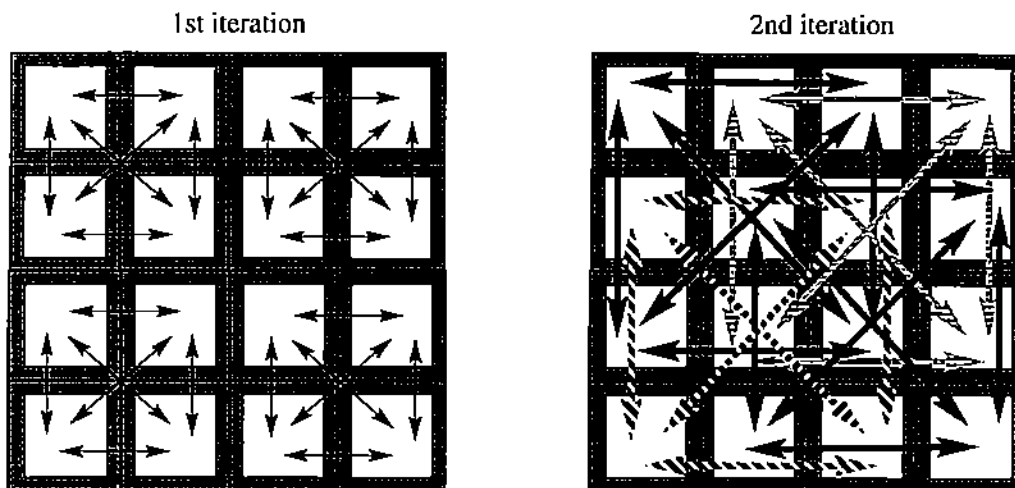


Figure 13: All-to-all broadcast patterns for Algorithm *logp-lev-quad*

our algorithms by the number of submachine levels they employ. Figure 12 contains an outline of three of the algorithms. In Algorithm *1-lev-dir* every processor sends its boundary list to every other processor. This is the only communication operation of the algorithm. After this communication, every processor can determine the head and rank in image I for every edge point on the boundary of its subimage. In Algorithm *2-lev-rc*, the processors first perform an all-to-all broadcast within every row, followed by an all-to-all broadcast within every column. The third algorithm, Algorithm *logp-lev-quad*, merges subimages in a quad-tree like fashion; i.e., at every iteration the boundary lists of four adjacent subimages are merged. Figure 13 shows the all-to-all patterns arising in the first two iterations of Algorithm *logp-lev-quad* on a 4×4 mesh. The processors communicating in the all-to-all broadcast in the second iteration are linked with arrows of the same type. In each one of these three contour ranking algorithms, every processor merges subimages at each iteration. At the same time, the number of processors merging identical subimages, and thus performing identical computations, increases with every iteration.

On a mesh architectures, Algorithm *logp-lev-quad* experiences the following communication imbalance. The size of the boundary of the subimages, and thus the size of the lists sent between processors, increases in subsequent iterations. In initial phases, processors communicate over short distances. As the algorithm proceeds, the communication distances and associated

congestion increases. This is also evident from Figure 13. This imbalance is the motivation for our fourth contour ranking algorithm, Algorithm *logp-lev-bal*. In Algorithm *logp-lev-bal* the imbalance is reduced by performing a permutation that sends the boundary list from processor $P_{i,j}$ to processor $P_{rev(i),rev(j)}$, where $rev(i)$ is the index obtained by applying the bit-reversal to the binary expansion of i. The result of applying this permutation is that processors initially communicate over long distances. As the size of the subimages and thus the sizes of the boundary lists increases, the distance between communicating processors and link congestion decreases.

5.3 Predicted Performance and Experimental Results

We next use the C^3-model to analyze the four contour ranking algorithm described in the previous section. Clearly, the performance of each one of the algorithms depends on the size of the boundary lists and is thus image-dependent. In order to analyze the algorithms, we need to make assumptions about the input. We measure for every image the edge point density which is defined as the fraction of all pixels that are edge points. We consider images with edge point densities from 5 to 100%. We use synthetic images consisting of vertical or diagonal lines through the entire image. The desired edge point density dictates the spacing of these lines. Real images with the same edge point density will give the same prediced performance and very similar experimental results.

We again use a 256-processor Intel Delta for our analysis. Figure 14 gives the communication and computation units of the algorithms for an image consisting of diagonal lines with an edge point density of 100%. The quantity B represents the size of the boundary list of subimage $I_{i,j}$ assigned to processor $P_{i,j}$.

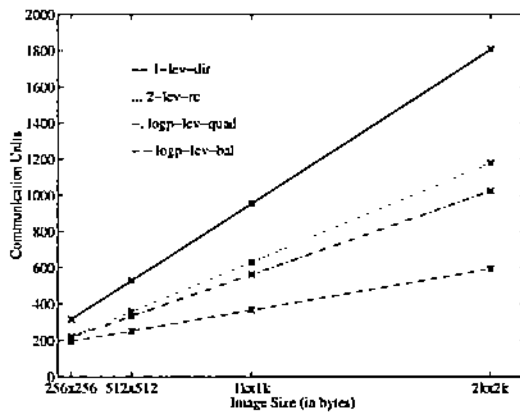
Figure 15(a) and (b) shows the predicted performance in terms of communication and computation units in graphical form for four image sizes ranging from 256×256 to $2K \times 2K$. Each image has an edge point density of 100%. The performance of the algorithms on a 256-processor Intel Delta is shown in Figure 15(c) and (d). Overall, Algorithm *logp-lev-quad* gave the best predicted and actual performance. We point out that each one of our algorithms experiences roughly $\log p$ message set-up costs. In Algorithms *1-lev-dir* and *2-lev-rc* these set-ups are experienced by the all-to-all broadcast (which is implemented using a binomial heap

Algorithm	Communication Units	Computation Units
1-lev-dir	$8s + 38 + 3.33B$	$1.5B$
2-lev-rc	$8s + 20 + 2.14B$	$0.84B$
logp-lev-quad	$8s + 38 + 1.8B$	$0.1B$
logp-lev-bal	$10s + 58 + 0.89B$	$0.11B$

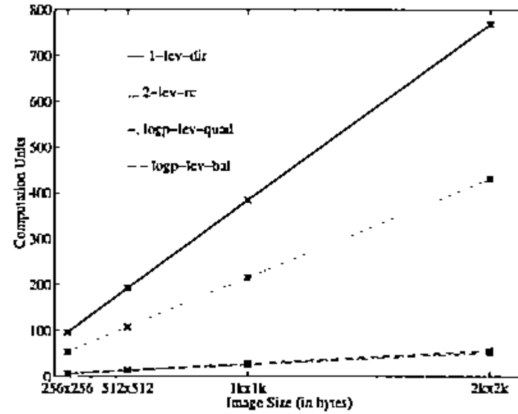
Figure 14: Approximate number of units charged for contour ranking algorithms on a 256-processor Intel Touchstone Delta with $h = 10$, $l = 512$, and $b = 16$, assuming nonblocking sends and receives and an edge density of 100%.

structure). In the quad-tree based algorithms, each of the $\log_4 p$ iterations experiences 2 set-up costs. The execution times reported in Figure 15(c) and (d) were obtained by monitoring one of the processors in the array. Since the edge density is uniform over the entire image, each processor experiences same computation and communication load. Therefor, monitoring a single processor not only gives a reasonable approximation of the overall performance, but also allows us to measure separetly the time spent in communication and on local computation. Comparison of the actual performance with the predicted performance for the algorithms reveals a high-degree of correlation between the two.

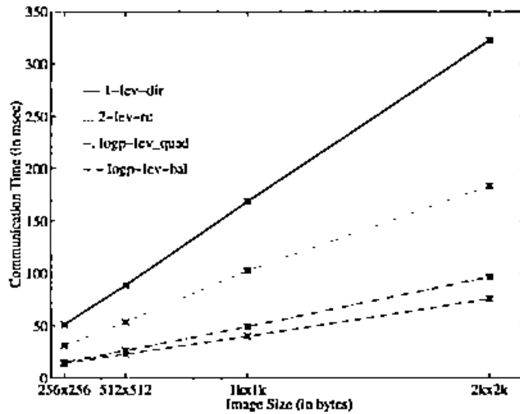
We conclude this section with a brief discussion on how the algorithms behave under different edge point densities. We considered edge point densities from 5 to 100%. The obtained results give insight into the behavior of the algorithms when the sizes of the boundary lists change. When increasing the edge point density in large images, we observed that the communication time of Algorithm *1-lev-dir* increases much sharper compared to the other algorithms. The growth rate in the communication time for Algorithms *logp-lev-quad* and *logp-lev-bal* is relatively slow. Algorithm *logp-lev-bal* gave the best performance for large images with a high edge-point density. This is attributed to the fact that for large, dense images the amount of data routed during the merging steps is significant enough to cause congestion in the routing network. Therefore, a data movement step before the actual merging in Algorithm *logp-lev-bal* pays off. For images of size $2K \times 2K$, Algorithm *logp-lev-bal* outperforms *logp-lev-quad* for images with an edge point density higher than 10%. Load balancing performs better by 10-15%. On the other hand, for images of size 256×256, the load balancing does not even pay for images with an



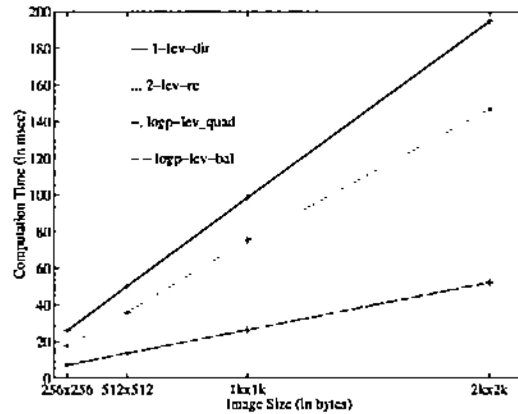
(a)



(b)



(c)



(d)

Figure 15: Predicted performance ((a) communication, (b) computation) and experimental results ((c) communication, (d) computation) of the contour ranking algorithms for images with an edge point density of 100% on a 256-processor Intel Delta.

edge point density close to 100%, as is also evident from Figure 15(c) and (d). The analysis of the actual performance on varying edge densities also conforms with the performance predicted by the C^3 model.

6 Conclusions

A computational model, the C^3 model, has been proposed for developing and analyzing algorithms on coarse-grained machines. The C^3 model allows evaluation of communication operations without a user having to specify fine scheduling details. Also, a metric has been defined to estimate the arising link and processor congestion. Coarse-grained algorithms have been developed for common communication operations and for a low-level vision problem solvable through divide-and-conquer algorithms. The validation of the model has been discussed by implementing the algorithms on the Intel Touchstone Delta and comparing the performance results with the predicted performance. This initial validation is encouraging and it provides insight into the interaction of various machine parameters and on their effect on the performance of coarse-grained algorithms.

7 Acknowledgements

We would like to thank Mike Atallah for helpful conversations and Farooq Hameed for his valuable assistance in the implementation of the algorithms.

References

[1] V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C.-T. Ho, S. Kipnis, and M. Snir, "CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers," *Proceedings of 8-th International Parallel Processing Symposium*, pp. 835-844, 1994.

[2] M. Barnett, R. Littlefield, D. G. Payne, and R. van de Geijn, "Global Combine on Meshes Architecures with Wormhole Routing," *Proceedings of 7-th International Parallel Processing Symposium*, pp. 156-162, 1993.

[3] A. Bar-Noy, S. Kipnis, "Designing Broadcasting Algorithms in the Postal Model for Message-Passing Systems," *Proceedings of 4-th ACM Symp. on Parallel Algorithms and Architectures*, pp. 13-22, 1992.

[4] S.H. Bokhari, "Multiphase Complete Exchange on a Circuit Switched Hypercube," *Proceedings of 1991 International Conference on Parallel Processing*, pp. 525-529, 1991.

[5] L. T. Chen, L. S. Davis, and C. P. Kruskal, "Efficient parallel processing of image contours," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 15, no. 1, pp. 69-81, 1993.

[6] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," *Proceedings of 4-th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, pp. 1-12, 1993.

[7] R. Cypher, E. Leu, "The Semantics of Blocking and Nonblocking Send and Receive Primitives," Technical Report, IBM Almaden Research Division, 1993.

[8] J.J. Dongarra, R. Hempel, A.J.G. Hey, D.W. Walker. "A Proposal for a User-level, Message Passing Interface in a Distributed Memory Environment", Technical Report TM 12231, Oak Ridge National Laboratory, 1993.

[9] C. Dwork, M. Herlihy, O. Waarts, "Contention in Shared Memory Algorithms", *Proc. of 25-th ACM STOC*, pp. 174-183, 1993.

[10] P.B. Gibbons, "A More Practical PRAM Model," *Proceedings of 1989 ACM Symposium on Parallel Algorithms and Architectures*, pp. 158-168, 1989.

[11] S.E. Hambrusch, F. Hameed, and A. Khokhar, "A Study of Coarse-Grained Communication Operations on Mesh Architectures" Technical Report, Purdue University, May 1994.

[12] F. Hameed, S.E. Hambrusch, A. Khokhar, and J.Patel, Contour Ranking on Coarse-Grained Machines: A Case Study for Low-level Vision Computations, Technical Report, Purdue University, November 1994.

[13] T. Heywood and S. Ranka, "A Practical Hierarchical Model of Parallel Computation: I. The Model," *Journal of Parallel and Distributed Computing*, Vol. 16, pp. 212-232, 1992.

[14] K. Hwang, *Advanced Computer Architecture with Parallel Programming*, McGraw-Hill, 1993.

[15] S.L. Johnsson, C.-T. Ho, "Optimum Broadcasting and Personalized Communication in Hypercubes," *IEEE Transactions on Computers*, Vol. 38, pp. 1249-1268, 1989.

[16] M.H. Kim, O.H. Ibarra, "Transformations Between Boundary Codes, Run Length Codes, and Linear Quadtrees," *Proceedings of the 8th International Parallel Processing Symposium*, pp. 120-125, 1994.

[17] P. Liu, W. Aiello, S. Bhatt, "An Atomic Model for Message Passing," *Proceedings of 5-th ACM Symp. on Parallel Algorithms and Architectures*, pp. 154-163, 1993.

[18] G. Papadopoulos, "Constant Factors Matter: Putting Communication on the Computation Power Curve," *Proceedings of DIMACS Workshop on Model, Architectures, and Technologies for Parallel Computation*, 1993.

[19] R. Ponnusamy, A. Choudhary, G. Fox, "Communication Overhead on CM5: An Experimental Performance Evaluation," *Proceedings of 4-th Symposium on the Frontiers of Massively Parallel Computation*, pp. 108-115, 1992.

[20] H. Samet, *Applications of Spatial Data Structures, Computer Graphics, and Image Processing*, Addison Wesley, 1990.

[21] D.S. Scott, "Efficient All-to-All Communication Patterns in Hypercube and Mesh Topologies," *Proceedings of 6-th Distributed Memory Computing Conference*, pp. 398-403, 1991.

[22] R. Thakur, A. Choudhary, "All-to-all Communication on Meshes with Wormhole Routing," *Proceedings of 8-th International Parallel Processing Symposium*, pp. 561-565, 1994.

[23] P. de la Torre and C.P. Kruskal, "Towards a Single Model of Efficient Computation in Real Parallel Machines," *Future Generation Computer Systems*, Vol. 8, pp. 395-408, 1992.

[24] L.G. Valiant, "A Bridging Model for Parallel Computation," *Communications of the ACM*, Vol. 33, No. 8, pp. 103-111, 1990.

[25] D.S. Wills and W. Dally, "Pi: A Parallel Architecture Interface," *Proceedings of 4-th Symposium on the Frontiers of Massively Parallel Computation*, pp. 345-352, 1992.