1994

Communication Operations on Coarse-Grained Mesh Architectures

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

Farooq Hameed

Ashfaq A. Khokhar

Report Number:
94-037

COMMUNICATION OPERATIONS ON COARSE-GRAINED MESH ARCHITECTURES

Susanne E. Hambrusch, Farooq Hameed and Ashfaq A. Khokhar
Computer Sciences Department
Purdue University
West Lafayette, IN 47907

CSD-TR-94-037
May, 1994

Communication Operations on Coarse-Grained Mesh Architectures *

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

Farooq Hameed
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA
hameed@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

May 19, 1994

Abstract

In this paper we consider three frequently arising communication operations, one-to-all, all-to-one, and all-to-all. We describe architecture-independent solutions for each operation, as well as solutions tailored towards the mesh architecture. We show how the relationship among the parameters of a parallel machine and the relationship of these parameters to the message size determines the best solution. We discuss performance and scalability issues of our solutions on the Intel Touchstone Delta. Our results show that in order to cover a broad range of scalability for a particular operation, multiple solutions should be employed.

Keywords: Parallel processing, coarse-grained machines, communication operations, scalability.

*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.

1 Introduction

Coarse-grained machines have emerged as major architectures in massively parallel computation. Achieving the speed-up these machines are capable of requires knowledge about the architectures, familiarity with the basic principles of parallel algorithm design, and an understanding of the impact of machine parameters on problem-solving approaches and implementations. Application programmers are not likely to be experts in all these areas. In order to improve the usability of parallel machines and to allow better utilization of high performance technology, implementations of fundamental operations should be fine-tuned to the hardware and software features of a particular machine. Communication operations are, without question, fundamental to parallel computation. Scalable and portable communication routines are the basis for making programs scalable and portable across different machines. Hence, it is important to understand the impact of architectural features and machine parameters on the performance of communication operations.

In this paper we consider one-to-all, all-to-one, and all-to-all communication. These three communication patterns arise in many applications and are a crucial component of a communications library. We describe different architecture-independent solutions, as well as solutions tailored towards the mesh architecture. We show how the relationship among the parameters of a parallel machine and the relationship of these parameters to the message size determines which solution is efficient in which environment. In addition to the number of processors and the message size, other parameters which influence performance include the cost of setting up a message, the ratio between send and receive times, the bandwidth of the processors and the network, the latency, the bisection width, and the type of synchronization used. Our conclusion is that for a given operation, different algorithms scale well for different ranges of input and different machine characteristics. This agrees with related work reported in [1, 2, 3, 4, 8, 11, 12]. We support our conclusion by presenting the performance of a number of diverse implementations for the Intel Touchstone Delta [10]. Some of our algorithms use well-know approaches, while others make use of characteristics intrinsic to the Intel Delta. We also address scalability issues and provide insight into the behavior of various algorithms on different machine sizes and data sizes.

Our algorithms assume that computation is synchronized by a barrier-style synchronization mechanism similar to the one described in [6, 14]. 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. 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. Hence, for a k-level algorithm, $k > 1$, to be efficient, the machine needs to support a limited use of process groups [5]. In our algorithms, processors belonging to the same process group form a scaled down version of the bigger machine. We thus refer to a process group as a submachine. Communication within different submachines occurs without interference. 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.

When describing our algorithms, we assume, for the sake of simplicity, that the size of the message routed between any two processors is L. 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. Our k-level algorithms are characterized by combining the original messages of size L and by performing independent routings within submachines. For all algorithms, the effective message size is never smaller than the actual message size.

In Section 2 we give a brief description of the Intel Delta. Section 3 discusses one-to-all communication, Section 4 all-to-one, and Section 5 all-to-all communication. In each section, we first discuss the different algorithms in an architecture-independent setting and then turn to the mesh architecture and the performance results achieved on the Intel Delta. For one-to-all we identify a $\log p$-level algorithm that performs well for all machine and message sizes we considered. All-to-one algorithms exhibit a different behavior than one-to-all algorithms. We identify a 2-level algorithm that performs reasonably well for all machine and message sizes. However, the choice of the best all-to-one algorithm for the Intel Delta depends on machine and message size. For all-to-all, our results clearly show that algorithms based on different approaches should be used for small and large message sizes, regardless of the size of the machine. We identify a 1-level algorithm which performs well for large message sizes and a 2-level algorithm which performs well for small message sizes.

2 Intel Delta

In this section we give a brief description of aspects of the Intel Delta relevant to the development of our algorithms and necessary for understanding the experimental results. For a more complete description we refer to [10].

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.

The machine sizes we considered in our experiments were 4×4, 4×8, 8×8, 8×16, and 16×16. The actual message sizes we considered varied from 16 to 16,384 bytes. Our code was written in C.

3 One-to-all Communication

In one-to-all communication a source processor P_s sends out $p-1$ distinct messages, each to a different destination. One-to-all is also refered to as the scatter or personalized broadcast operation [5, 8]. The source processor is clearly the bottleneck. In Section 3.1 we use the concept of a k-level algorithm to describe different algorithms. The amount of data sent out by the source processor is the same for all algorithms, but the algorithms differ on how the actual messages are combined into larger messages which are sent to their destinations via intermediate processors. The objective of all algorithms is (i) to have processor P_s send out the $p-1$ messages as fast as possible and (ii) to minimize the time between processor P_s sending out the last packet and a processor receiving the last packet of its message. How to best minimize this time difference depends on the message size and features of the underlying machine. Section 3.2 discusses performance and scalability issues for the Intel Delta.

3.1 The Algorithms

There exist two conceptually different 1-level algorithms for one-to-all communication. One approach is to have processor P_s issue $p-1$ direct sends (and every other processor issues a receive). This strategy is likely to be used by a programmer not familiar with parallel processing and it is likely to perform well on small machines (fewer than 16 processors). Another approach is to have processor P_s form one long message of size $L(p-1)$ which is broadcast to every processor (i.e., the effective message size is $L(p-1)$). After receiving this message, every processor extracts the message destined for it. One expects the broadcasting approach to be efficient only when L is small and when the parallel machine has a control network dedicated to fast broadcasts.

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. Processor P_s 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, $0 \leq i < p^\alpha$. Next, processor P_s issues p^α sends (or $p^\alpha - 1$ sends if P_s is a leader) to route the long messages to the leaders. Once a leader received its long message, it initiates a 1-level one-to-all algorithm within its submachine. A 3-level algorithm is obtained by applying the above 2-level approach to each submachine.

An interesting class of algorithms arises when each superstep partitions into two submachines and the number of processors in each submachine is a fraction of the original number. We call such an algorithm a Binomial Heap algorithm (since the sends issued induce a tree having the shape of a binomial heap) and also refer to it as a $\log p$-level algorithm (the number of supersteps is proportional to $\log p$). When the machine is divided into submachines of equal size in each superstep, we perform $\log p$ supersteps and minimize the total number of message set-up costs experienced.

The approaches described above are architecture-independent. For most existing architectures, there exist 2-, 3-, and $\log p$-level algorithms that minimize link congestion and that allow the lower level algorithms to be executed on independent submachines. For the mesh architecture, submachines consisting of a row or a column or submachines having the same aspect ratio as the original mesh are natural choices. We conclude this section by describing two $\log p$-level algorithms especially suitable for the mesh. A natural approach for a 2-dimensional mesh is to alternate making vertical and horizontal cuts. For a square p-processor mesh, the algorithm operates then on a square mesh of size $p/4$ after two supersteps. Another approach is to divide the mesh into two submachines based on a given parameter γ, $0.5 \leq \gamma < 1$. The division is made so that the submachine containing the source processor P_s consists of γp processors and the other submachine contains the remaining $(1-\gamma)p$ processors. The motivation for partitioning into two submachines of different size comes from our experience with the Intel Delta on which processors can send data faster than they can receive it. Clearly, since data cannot be received faster than it is sent out, a value of $\gamma < 0.5$ cannot give a better performance for one-to-all communication.

3.2 Implementations and Experimental Results

In this section we describe different one-to-all algorithms we implemented on the Delta, and discuss their performance and related scalability issues. For clarity, an outline of the algorithms is given in Figure 1. The actual implementations handle rectangular meshes, but for simplicity the algorithms are stated in the outline for square meshes. When a processor issues multiple sends, our implementations give higher priority to destinations further away. We use this simple rule to increase the amount of possible pipelining, minimize congestion, and minimize the time between P_s sending its last message and a processor receiving its message from P_s. Since we assume that each message has length L, we did not run into the situation in which messages sent out by a processor have different length. In such a case, longer messages should be given preference over shorter ones.

We considered three 1-level algorithms: Algorithm *1-lev-dir*, which issues direct sends, Algorithm *1-lev-sys-br*, which uses the system's broadcast, and Algorithm *1-lev-our-br*, which uses a broadcasting tree in the form of a binomial heap. We implemented one 2-level algorithm, Algorithm *2-lev-rec*, in which each submachine consists of a row of processors. The leaders are the processors in the same column as processor P_s. We use Algorithm *1-lev-dir* as the 1-level

Algorithm 1-lev-dir(p)

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

Algorithm 1-lev-sys-br(p)/1-lev-our-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 1: Outline of one-to-all algorithms implemented on the Intel Delta.

algorithm within each row. Algorithm *3-lev-sq* is a 3-level algorithm. For square mesh sizes, the p-processor machine is logically partitioned into $\sqrt{p}$ submachines, each being an array of size $p^{1/4} \times p^{1/4}$. If the source processor is in row i and column j of a submachine, then the processor in row i and column j of each submachine is the leader in its submachine. This convention avoids sending data from P_s to another processor in the same submachine. Once a leader receives its long message from P_s, it initiates a 2-level algorithm using Algorithm *2-lev-rec* within its submachine.

Algorithm *logp-lev-sq* is the $\log p$-level algorithm alternating vertical and horizontal cuts. Algorithm *logp-lev-rec*(γ) is an algorithm partitioning the machine into two submachines using γ as the partitioning factor, $0.5 \leq \gamma < 1$. The partitioning is done by viewing the processors as being indexed in snake-like row-major order. Let s be the index of the source processor in this indexing schema. If $s < \gamma p$, we assign the γp processors with smallest index to one submachine (and the remaining $(1-\gamma)p$ processors to the second submachine). If $s \geq \gamma p$, we assign the γp processors with largest index to one submachine. Observe that for $\gamma = 0.5$ and p a power of two, we perform $\log p$ supersteps. If the mesh is square, the first half of the supersteps can be viewed as making horizontal cuts and the second half making vertical cuts.

The experimental results of the one-to-all algorithms obtained from a 256-processor Intel Delta for $P_s = P_0$ are shown in Figure 2. We chose processor P_0 as the source since it gives a worse performance than a source processor more in the center of the mesh. We give the performance of the algorithms using nonblocking sends. The performance using blocking sends is consistently worse for one-to-all routing.

For all machine sizes considered (which ranged from 16 to 256 processors), the relative performance of the algorithms was the same. Hence, the following discussion applies to all machine sizes. Algorithm *1-lev-dir* minimizes the effective message size, but experiences a total of $p-1$ message set-up costs. *1-lev-dir* is a reasonable choice only for large message sizes (at least 4 Kbytes). We point out that sending messages of size ≤ 482 bytes costs approximately the same. The two broadcasting algorithms, Algorithms *1-lev-sys-br* and *1-lev-our br*, give the worst performance of all algorithms, with the system's broadcast performing significantly worse than our own broadcast. Because of the poor expected performance of *1-lev-sys-br*, we did not

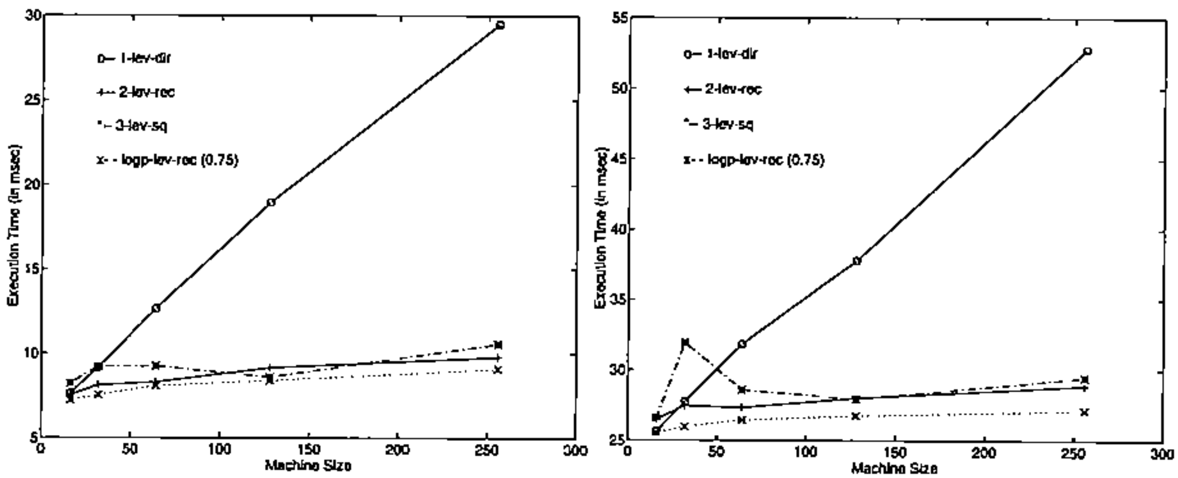
One-to-All Algorithms	Message Size (in Bytes)										
	16384	8192	4096	2048	1024	512	256	128	64	32	16
1-lev-dir	420.82	226.21	130.80	75.22	52.80	37.61	29.50	28.58	26.05	26.45	26.04
1-lev-sys-br	–	–	–	1773.89	887.07	442.78	219.30	110.68	53.65	27.43	13.13
1-lev-our-br	4377.44	2193.51	1096.23	549.12	275.79	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	4.48	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 2: Performance results for one-to-all communication on a 256-Processor Intel Delta (times are in msec).

run this algorithms on messages sizes of 4 Kbytes and more. The poor performance is partly due to the large effective message size (it remains Lp throughout), as well as due to the absence of a dedicated fast broadcasting network in the Delta.

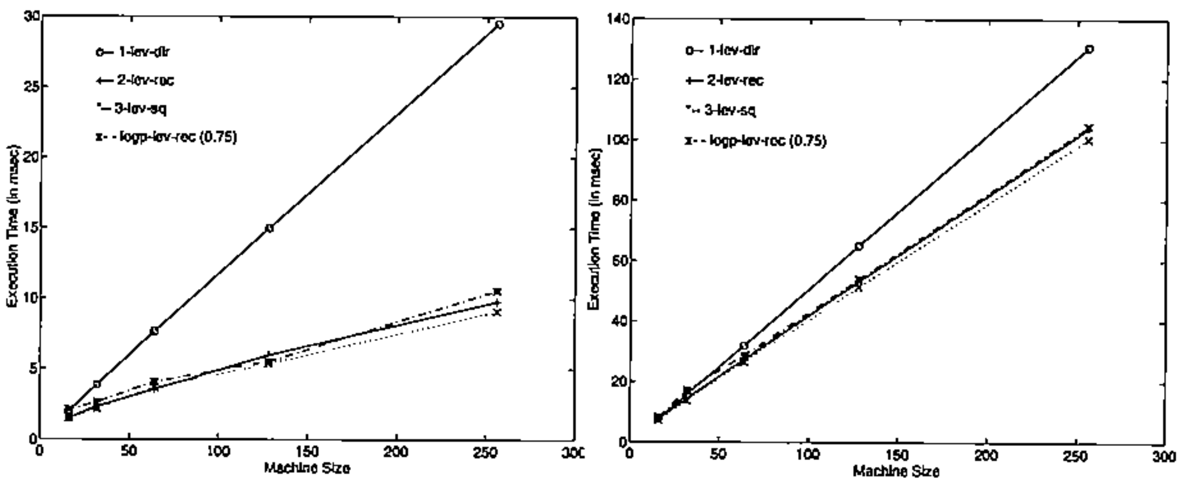
Of all the algorithms, *2-lev-rec*, *3-lev-sq* and *logp-lev-rec*(0.75) perform the best. This holds for all message and machine sizes, with the exception of *3-lev-sq* for small machine sizes. We believe that Algorithm *logp-lev-rec*(0.75) performs well because it is tailored towards the Delta. The value of $\gamma = 0.75$ was obtained through experiments. This value gave optimal or near optimal results for all machine and message sizes. As one would expect, Algorithm *logp-lev-sq* and Algorithm *logp-lev-rec*(0.5) give about the same performance. Algorithm *3-lev-sq* balances the effective message size, the number of messages sent, and the bisection width of the underlying submachines more than any of the other algorithm. We expect that on a mesh architectures in which a processor can send out data via different links simultaneously, the performance of this 3-level algorithms compared to *2-lev-rec* and, *logp-lev-rec*(0.75) would improve.

Figures 3(a) and 3(b) show the scalabilty behavior of four one-to-all algorithms when the total number of bytes sent out by the source processor is 64 Kbytes and 256 Kbytes, respectively, and the machine size varies from 16 to 256 processors. This corresponds to the situation when the problem size remains constant. Algorithms *2-lev-rec*, *3-lev-sq* and *logp-lev-rec*(0.75) show



(a) (b)

Figure 3: Scalability results for processor P_s sending a total of 64 Kbytes and 256 Kbytes, respectively, varying machine size.



(a) (b)

Figure 4: Scalability results for processor P_s sending an actual message size of 256 bytes and 4 Kbytes, varying machine size.

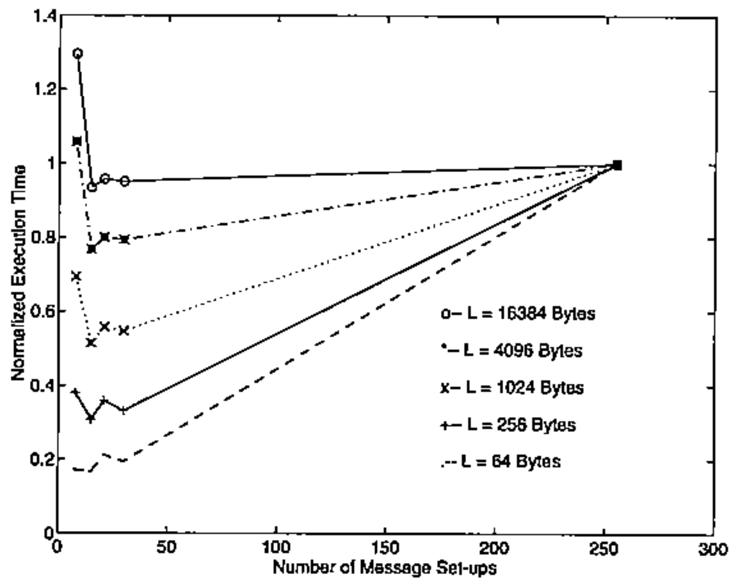


Figure 5: Scalability results for five one-to-all algorithms, showing the number of message set-ups on a 256-processors machine, varying the actual message size.

an ideal behavior. For small meshes (e.g., 4×8), the advantages of the 3-level algorithm are almost lost. This shows up in the graphs, especially in Figure 3(b). Figure 4 shows the scalabilty behavior of the same four one-to-all algorithms when the actual message sizes are 256 bytes and 4 Kbytes, respectively. Again, Algorithms *2-lev-rec*, *3-lev-sq* and *logp-lev-rec*(0.75) show an ideal behavior. In comparison to Algorithm *1-lev-dir*, these algorithms performs well for small messages size while the performance gap narrows for messages of size 4 Kbytes.

Figure 5 provides insight into the relationship between the total number of message set-ups experienced and the actual message size. Observe that for a fixed machine size, the number of message set-ups experienced by an algorithm does not change as the message size increases. For a 256-processor machine, Algorithm *logp-lev-sq* experiences 8, *logp-lev-rec*(0.75) experiences 15, *3-lev-sq* 21, *2-lev-rec* 30, and *1-lev-dir* experiences 255 message set-up costs. For better illustration, we normalized the execution time to the time taken by Algorithm *1-lev-dir*. The figure demonstrates in an interesting way the effect of the number of message set-ups on the overall performance. With the increase in message size, the effect of the set-up cost on the overall performance decreases for all algorithms. This can be observed by the almost flat line for $L = 16,384$.

In summary, our experimental work on the Intel Delta indicates that the message-combining algorithms (excluding the broadcasting algorithms) perform well for small message sizes; i.e., when $L \leq 256$ bytes. For large message sizes, Algorithm *logp-lev-rec*(0.75) is the best choice, independent of the machine size. We expect our message-combining algorithms to perform well for small messages on other architectures as well. Which one of them gives the best performance will depend on the ratio between the send and receive time, the packet length, the ratio between the processor and network bandwidth, and the start-up cost.

4 All-to-one Communication

In all-to-one communication, also known as the gather operation [5], every processor sends a message to a destination processor, P_d. Processor P_d is now the bottleneck. Conceptually, all-to-one is the inverse of one-to-all. However, from a practical point of view, the best one-to-all algorithms do not necessarily correspond to the best all-to-one algorithms. In this section we describe different all-to-one implementations and discuss their performance on the Intel Delta. We then compare the performance of all-to-one algorithms to that of one-to-all's.

All one-to-all algorithms, except the algorithms based on broadcasting, have corresponding all-to-one algorithms. Algorithm *1-lev-dir* for all-to-one is an implementation in which every processor issues a send to processor P_d (and P_d issues $p-1$ receives). 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, $0 < \gamma < 1$. For one-to-all, the assumption $\gamma \geq 0.5$ guarantees that the source processor P_s is in the submachine containing γp processors. For all-to-one, allowing $\gamma < 0.5$ can create the following scenario: When determining the submachines based on their snake-like row-major index, neither of the first γp, nor the last γp processors may now contain the destination processor P_d. In this situation (i.e., $\gamma p < d < (1-\gamma)p$), Algorithm *logp-lev-rec*(γ) uses d to partition into submachines. The partition is chosen so that one submachine contains the first $d-1$ processors and executes an all-to-one communication with P_{d-1} as destination. The remaining processors belong to the second submachine and they continue with P_d as the destination. It is easy to see that for $\gamma < 0.5$ such a partition around the destination processor occurs at most once during the algorithm. For the Delta, we did not expect a value of $\gamma < 0.5$ to give a better performance and experimental work has confirmed this.

4.1 Implementations and Experimental Results

The experimental results for the all-to-one algorithms obtained from a 256-processor Intel Delta for $P_d = P_0$ using nonblocking sends are shown in Figure 6. The performance using blocking

All-to-One Algorithms	Message Size (in Bytes)										
	16384	8192	4096	2048	1024	512	256	128	64	32	16
1-lev-dir	1451.35	540.16	230.57	118.24	66.65	40.26	22.76	20.06	16.37	16.76	16.70
2-lev-rec	564.30	287.27	145.79	95.32	31.75	17.07	9.40	5.99	4.01	3.14	2.54
3-lev-sq	610.34	300.63	140.16	67.60	37.79	18.98	10.91	6.41	4.22	3.13	2.65
logp-lev-sq	548.38	276.10	139.49	71.04	37.10	19.75	11.18	6.78	4.73	3.42	2.85
logp-lev-rec(0.60)	508.84	255.51	128.98	67.53	35.17	18.93	10.70	6.30	4.35	3.25	2.91

Figure 6: Performance results for all-to-one communication on a 256-Processor Intel Delta (times are in msec).

sends was consistently worse, with the exception of Algorithm *1-lev-dir*. Algorithm *1-lev-dir* using blocking sends performed 4-10 msec better than *1-lev-dir* using nonblocking sends (the exact value depends on the message size). However, for all machine and all message sizes we considered, the performance of Algorithm *1-lev-dir* does not come close to that of the better performing algorithms. 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. Observe that for Algorithm *2-lev-rec* the table shows a 3-fold increase in time when the message size doubles from 1024 to 2048. We observed such an undesirable behavior in more than one all-to-one algorithm. In particular, it showed up for certain values of γ in Algorithm *logp-lev-rec*(γ) when $L = 2048$ to $L = 4096$. We are not able to provide an explanation, but it appears that some system limits are being exceeded.

Overall, for all message and all machine sizes we considered, Algorithm *2-lev-rec* is a good choice. Its good performance is also apparent from Figures 7 and 8. However, Algorithm *2-*

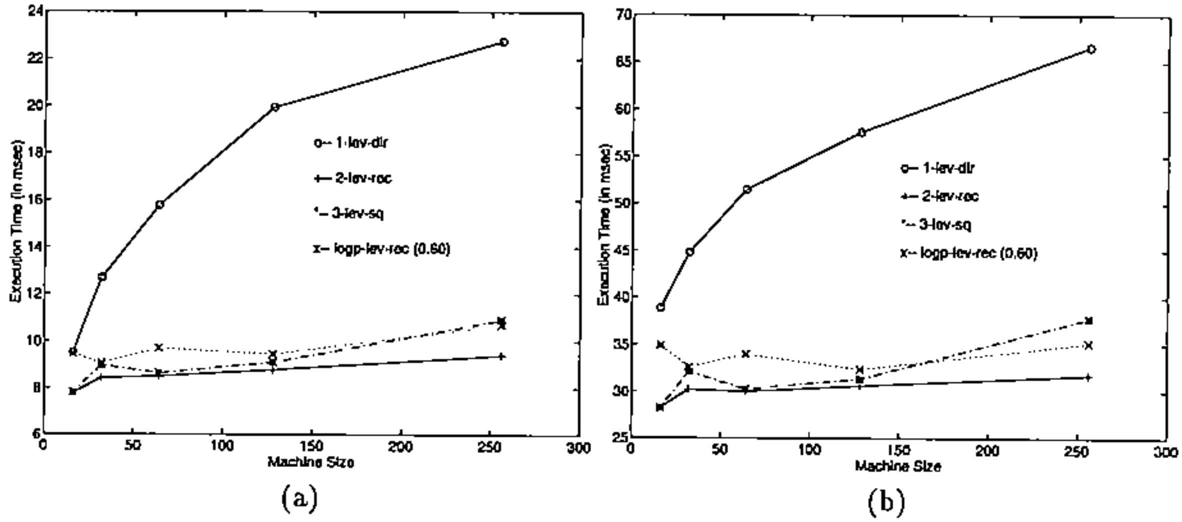
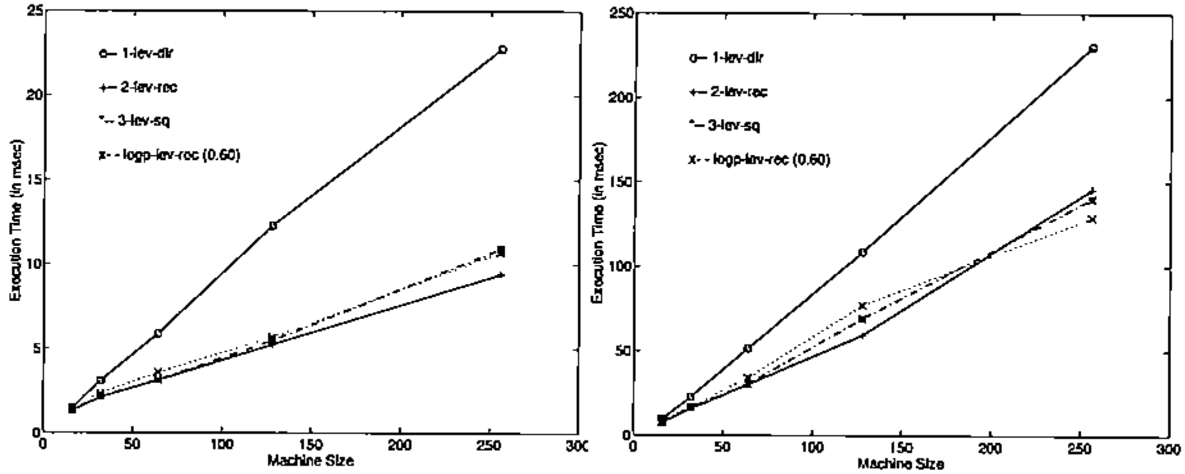


Figure 7: Scalability results for processor P_d receiving a total of 64 Kbytes and 256 Kbytes, respectively, varying machine size.

lev-rec does not provide the best results for all machine size/message size pairs. Figures 7 and 8 show the scalability behavior of four algorithms, Algorithm *1-lev-dir*, *2-lev-rec*, *3-lev-sq*, and *logp-lev-rec*(0.60). In the first figure we keep the problem size fixed and in the second one the message size, varying the machine size in both cases.

For one-to-all we found that $\gamma = 0.75$ gave an optimal or near optimal performance for all machine and message sizes. For all-to-one, we cannot identify a single value of γ that gives a good performance. For each machine size, a different range of γ's worked best. In addition, for a fixed machine size, the message size influenced the choice of γ. For example, for a 256-processor machine, $0.60 \leq \gamma \leq 0.65$ performs well, with $\gamma = 0.65$ performing better for $L \leq 2048$ bytes and $\gamma = 0.60$ performing better for $L > 2048$ bytes. Using $\gamma = 0.65$ for large messages increased the time by about 40%. The pattern of a slightly larger value of γ giving a better performance for messages of size ≤ 2048 bytes and a smaller value of γ giving a better performance messages of length more than 2048 bytes holds for all machine sizes we considered. For example, for 16- and 32-processor machines the γ-values are 0.70 and 0.60.

Comparing the performance of the all-to-one to the one-to-all algorithms provides interesting insight into how machine parameters can influence performance. Recall that the all-to-one algorithms correspond to one-to-all algorithms with the role of the sends and receives reversed.



(a) (b)

Figure 8: Scalability results for processor P_d receiving actual message sizes of 256 bytes and 4 Kbytes, varying machine size.

However, for messages of length < 512 bytes the all-to-one algorithms are slightly faster than their one-to-all counterpart, while for messages of length ≥ 512 bytes the all-to-one algorithms are significantly slower. This can be explained as follows. In our algorithms for one-to-all, when a processor issues multiple sends, higher priority is given to the destinations further away. For all-to-one, when a processor issues multiple receives, the processors are not able to employ such a rule. In addition, a processor issuing multiple receives experiences an additional overhead when dealing with the arbitrary arrival of messages and determining which posted receive corresponds to an arriving message. Finally, for the Delta the set-up time of receiving a message is less than the set-up time of sending a message, while a processor can send out data faster than it can be received.

Using the above observations, one would expect Algorithm *logp-lev-sq* to exhibit a similar performance for one-to-all and all-to-one. Our results for the Delta support this statement (compare 2nd last row of Figures 6 and 2). (Recall that in *logp-lev-sq* every processor issues at most one send and at most one receive in a superstep.) For small message sizes, the set-up cost experienced when sending messages constitutes a bigger fraction of the overall time. Since the set-up time of receiving a message is less than the set-up time for sending, it is not surprising that most all-to-one algorithms are faster than their one-to-all counterpart for small messages.

For large message sizes, the one-to-all algorithms are faster. Contrary to one-to-all, all-to-one algorithms are not able to effectively use the buffering capacity of the network to exploit the difference in send and receive rates. This appears to be the main reason for the increase in time on the Delta.

5 All-to-all Communication

In all-to-all communication every processor sends a distinct message to every other processor. When performing an all-to-all, the congestion arising because of the bisection of the underlying architecture can significantly influence the performance. The *bisection width* of a machine is the minimum number of links that have to be removed to disconnect the machine into two equal-sized halves [9]. In a p-processor architecture with a bisection width of b, at least one of the b links partitioning the machine is used by at least $p^2/4b$ messages during an all-to-all communication. Thus algorithms for all-to-all not only have to consider how to combine actual messages into larger messages, but they have to address how congestion can be avoided or is handled. In Sections 5.1 and 5.2 we describe a number of 1-level algorithms and discuss higher level algorithms, respectively. Section 5.3 discusses the performance of different implementations based on these algorithms on the Intel Delta.

5.1 1-level Algorithms

The most straightforward 1-level approach is the one in which each processor sends its $p-1$ messages, one by one, regardless of what the other processors are doing. In such an algorithm no combining of messages is done, the machine is flooded with messages, and the arising congestion is left to be handled by the system.

A frequently used approach that attempts to control congestion implements all-to-all through $p-1$ or p one-to-one routings. More precisely, the $p(p-1)$ message routing requests are partitioned into permutations. We view such algorithms as 1-level algorithms. Common partitioning schemas are linear permutations and exclusive-or permutations. When partitioning into *linear* permutations, processor P_j sends a message to processor $P_{(i+j)\,\mathrm{mod}(p-1)}$ in the i-th permutation, $1 \leq i \leq p-1$. When partitioning into *exclusive-or* permutations, all-to-all is partitioned such that in the i-th permutation processor P_j sends a message to $P_{i \oplus j}$. Implementations of these approaches on different architectures have shown exclusive-or permutations to be superior to linear permutations [11, 12].

In order to evaluate different partitioning schemas, we define two quantities, *max_load* and *sum_load.* Assume all-to-all is partitioned into p permutations, $\Pi_0, \ldots \Pi_{p-1}$. The load of a link in permutation Π_i is defined as the number of messages using this link in the same direction during the routing of permutation Π_i. The load of permutation Π_i, $load(\Pi_i)$, is defined as the maximum load over all links during the routing of permutation Π_i. Let $max\_load = \max_{0 \le i \le p-1} load(\Pi_i)$ and $sum\_load = \sum_{i=0}^{i=p-1} load(\Pi_i)$.

Consider a p-processor square mesh architecture with $\sqrt{p}$ being a multiple of 4. Any partitioning into permutations gives $max\_load \ge \sqrt{p}/4$ and $sum\_load \ge p^{3/2}/4$ [13]. Linear and exclusive-or permutation have $max\_load = \sqrt{p}/2$, which is a factor of 2 off from the optimal *max_load.* For exclusive-or permutations we have $sum\_load = \frac{3}{7}p^{3/2}$, which is a factor of 12/7 off from the optimal *sum_load.* Using an approach developed in [13], all-to-all communication can be partitioned into p permutations achieving $max\_load = \sqrt{p}/4$ and $sum\_load \ge p^{3/2}/4$. We refer to this approach as partitioning into *balanced* permutations. For completeness sake, we describe the method given in [13] for generating balanced permutations. We start by describing balanced permutations for linear arrays. The permutations for the mesh are obtained by performing a cross product.

Consider a k-processor linear array. Assume, for the time being, that k is a multiple of 4. Logically partition the linear array into a left half and into a right half. Next, determine a tournament involving $k/2$ "players". Such a tournament consists of $k/2 - 1$ rounds, where in each round one player is matched up with exactly one other player. The rounds can be generated by using, for example, the method given in [7] for finding the 1-factors of a complete graph. Assume i is matched up with j in a round, $0 \le i < j < k/2$. Then, the cycle

$$P_i \rightarrow P_j \rightarrow P_{k-i-1} \rightarrow P_{k-j-1} \rightarrow P_i$$

describes the sending of four messages. Hence, the $k/2$ match-ups of round induce two permutations (in the second permutation we simply interchange sending and receiving processors). From the $k/2-1$ rounds of a tournament we obtain a total of $k-2$ permutations. The messages that remain to be sent are the ones in which processor P_i sends and receives from processor P_{k-i-1}, $0 \leq i < k/2$. In order to achieve $max\_load = k/4$, these final messages are routed in two permutations, resulting in a total of k permutations. It is easy to see that each of the k permutations has a load of $k/4$, giving $max\_load = k/4$ and $sum\_load = k^3/4$.

We briefly comment on how to handle values of k that are not a multiple of 4. Assume first $k = 4i+2$. We introduce one "dummy player" in the tournament, resulting in $2i+2$ tournament players. All-to-all can now be done in k permutations, with half the permutations having a load of $\lceil k/4 \rceil$ and half having a load of $\lfloor k/4 \rfloor$. When $k = 4i+3$, all-to-all can be partitioned into k permutations, with each permutation having a load of $\lceil k/4 \rceil$. Finally, for $k = 4i+1$, the approach of creating dummy players results in $k+1$ permutations, half having a load of $\lceil k/4 \rceil$ and half having a load of $\lfloor k/4 \rfloor$.

Consider now a 2-dimensional p-processor mesh with $p = r \cdot c$. Let $\Pi_0, \ldots, \Pi_{r-1}$ be the r balanced permutations of an r-processor linear array, and let $\Pi'_0, \ldots, \Pi'_{c-1}$ be the c balanced permutations of a c-processor linear array. Then, $\Pi_i \times \Pi'_j$ gives the p balanced permutations with $max\_load = max\{\lceil r/4 \rceil, \lceil c/4 \rceil\}$.

In summary, we have described three partitioning approaches that can be implemented on any p-processor architectures supporting one-to-one communication. For a mesh architecture, partitioning into balanced permutations is optimal with respect to the defined quantities measuring link congestion. All three partitioning approaches can be applied to 2-dimensional meshes of any size.

5.2 Higher-level Algorithms

In this section we describe two 2-level algorithms and one commonly used $\log p$-level algorithm. The 2-level algorithms can be generalized to higher-level algorithms. For $k > 1$, a k-level algorithm combines the actual messages into larger messages, with the goal of achieving a better performance for smaller message sizes. To simplify the description of the algorithms, we assume a square mesh of size $\sqrt{p} \times \sqrt{p}$. In the 2-level algorithms, the p-processor machine is logically partitioned into $\sqrt{p}$ submachines, $S_0, \ldots S_{\sqrt{p}-1}$.

We start with the description of the first 2-level algorithm. It consists of 3 steps and we refer to it as the 3-step algorithm. A similar approach for hypercube architectures has been described in [3]. In each step of the algorithm every processor sends out a total of pL bytes; the first and the last step send out pL bytes in the form of $\sqrt{p}$ messages and the second step sends them out as one single message. The goal of the first step is to have processor P_i in submachine S_j contain the p messages originating within submachine S_j and destined for the processors in submachine S_i. This is achieved by performing a 1-level all-to-all algorithm within each submachine. The length of the message sent from processor P_k to processor P_i in S_j is $\sqrt{p}L$. The second step is a one-to-one communication. Processor P_i of submachine S_j sends a concatenation of the $\sqrt{p}$ messages it received in the first step to processor P_j in submachine S_i. The communication pattern of this one-to-one operation has the flavor of a transpose and, depending on the architecture, it could be congestion-prone. The third and final step is again an all-to-all communication within each submachine. The message of size pL received in the second step is partitioned into $\sqrt{p}$ equal-sized messages, each one destined for a different processor in the submachine. After this all-to-all communication, every processor contains the $p-1$ messages destined for it.

Our second 2-level algorithm consists of only 2 steps, with each step sending out a total of pL bytes in the form of $\sqrt{p}$ messages. We refer to it as the 2-step algorithm. The potential disadvantage of this algorithm is the requirement that each one of the two steps needs a different submachines partitioning with the following property. Let $S_0, \ldots S_{\sqrt{p}-1}$ be the partition used in the first step and let $T_0, \ldots T_{\sqrt{p}-1}$ be the one used in the second step. Then, there exists exactly one processor, say P_{ij}, that is in submachine S_i and in submachine T_j, $0 \leq i, j \leq \sqrt{p}-1$. The first step performs an all-to-all communication within each submachine S_i so that P_{ij} contains the p messages to be sent from processors in S_i to processors in T_j. The second step performs an all-to-all communication to delivers the messages at their final destinations within each submachine T_j.

Finally, consider the following $\log p$-level algorithm which is based 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. This approach has consistently been judged as being expensive for large message sizes [3, 12].

5.3 Implementations and Experimental Results

We have implemented a total of eight all-to-all algorithms on the Delta. This includes four 1-level algorithms: Algorithm *1-lev-dir*, in which each processor simply issues its $p-1$ sends and $p-1$ receives, and three algorithms that partition all-to-all communication into permutations. These algorithms are Algorithm *1-lev-lin*, *1-lev-xor*, and *1-lev-bal* and they partition into linear, exclusive-or, and balanced permutations, respectively.

We have implemented three 2-level algorithms. Algorithm *2-lev-sq* corresponds to the 3-step algorithm described in the Section 5.2. For a square mesh, each submachine is a square submesh of size $p^{1/4} \times p^{1/4}$. Algorithm *2-lev-c,r* corresponds to the 2-step algorithm described in Section 5.2. In this algorithm submachine S_i corresponds to the i-th column and submachine T_j corresponds to the j-th row of the mesh. We use Algorithm *1-lev-xor* as the 1-level algorithm within the columns (and then the rows). For the sake of comparison, we also considered a variation of Algorithm *2-lev-c,r* reported in [12]. The differences are as follows. As before, a processor in S_i sends the corresponding data to processor P_{ij}. However, processor P_{ij} does not wait until all $\sqrt{p}$ large messages have been received, but sends out messages of size L to the destination processors in submachine T_j as soon as they are received. This interleaves the two steps and we refer to the corresponding algorithm as Algorithm *2-lev-c,r-int*. The 8-th algorithm is Algorithm *logp-lev-bfly* which uses the butterfly communication pattern.

Figure 9 shows the performance of these eight algorithms on a 256-processor Delta, varying the message size from 16 bytes to 16,384 bytes. We only report the performance for non-blocking sends (use of blocking sends increased the time). Algorithm *1-lev-xor* gives the best performance for larger message sizes; i.e., $L \geq 256$. For small messages sizes (i.e., $L \leq 256$), Algorithm *2-lev-c,r* achieved the best performance. This conclusion holds not only for a 256-processors machine, but for all machine sizes we considered. Figure 10 shows the scalabilty behavior of Algorithms *1-lev-dir*, *1-lev-xor*, *2-lev-c,r*, *2-lev-c,r-int*, and *2-lev-sq* with actual message sizes of 64 bytes and 4096 bytes, respectively, varying over different machine sizes.

We briefly comment on the performance of the other algorithms compared to *1-lev-xor* and *2-lev-c,r*. As expected, Algorithm *1-lev-lin* did consistently worse than *1-lev-xor*. We found the

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 9: Performance results for all-to-all communication on a 256-processor Intel Delta (times are in msec).

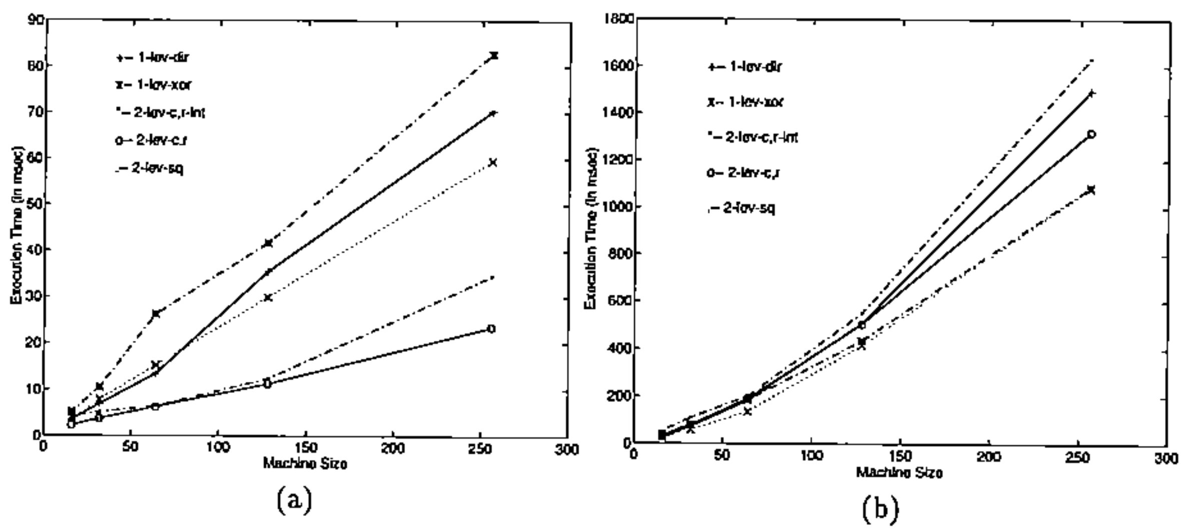


Figure 10: Scalability results for all-to-all algorithms with actual message sizes of 64 bytes and 4 Kbytes, varying machine size.

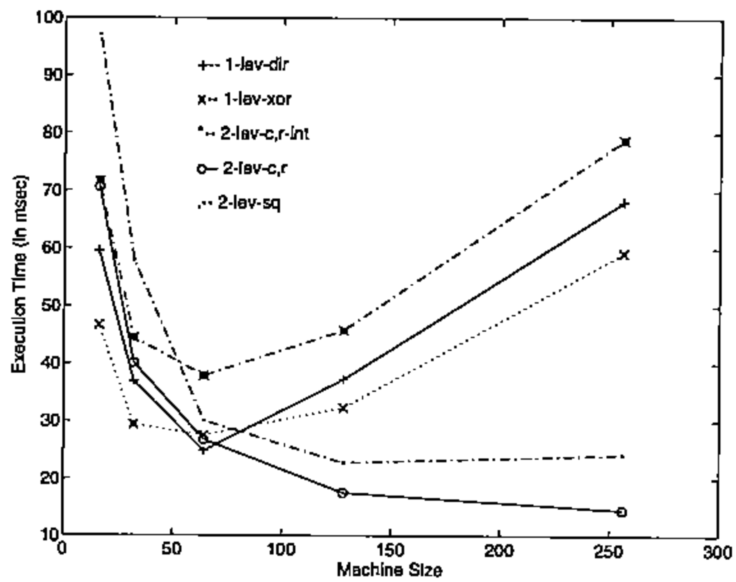


Figure 11: Scalability results for all-to-all algorithms when the total of all actual messages is 2 Mbytes, varying machine size.

performance of Algorithm *1-lev-bal* on the Delta disappointing. The advantage of partitioning into balanced permutations compared to exclusive-or permutations did not show up on the Delta. We conjecture that for other mesh architectures Algorithm *1-lev-bal* could be superior. 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 may show up for larger machine sizes when the small bisection width of the submachines used in the 2-step algorithm starts to influences performance. Algorithm *2-lev-c,r-int* outperforms *2-lev-c,r* only for larger machine sizes ($p \geq 64$) and larger message sizes ($L \geq 1024$ bytes).

We conclude this section by showing in Figure 11 the scalabilty behavior of five all-to-all algorithms when the total of the actual messages sent between all processors is 2 Mbytes. This corresponds to keeping the problem size fixed and changing the machine size. For a 256-processor machine, every processor sends an actual message of 32 bytes to every other processor. Figure 11 clearly indicates that an efficient and scalable all-to-all implementation should employ different algorithms for large and small message sizes. For a total message size of 2 Mbytes, the switch away from an algorithm that combines messages occurs when a 64-processor machine sends messages of size 512 bytes. This is the point at which Algorithm *1-lev-xor* starts to outperform Algorithm *2-lev-c,r*.

6 Conclusions

We have presented several architecture-independent algorithms for one-to-all, all-to-one, and all-to-all communication, as well as algorithms tailored towards mesh architectures. In addition to using the concept of a k-level algorithm, our solutions can be characterized by the maximum number of sends/receives issued by a processor and the sizes of messages exchanged among processors. The proposed algorithms have been implemented on the Intel Delta and performance results were shown. We discussed the behavior of the algorithms on different machine sizes over a broad range of message sizes. Our conclusion is that for a given operation, different algorithms scale well for different ranges of input and machine size. We have supported this conclusion by presenting the performance of diverse and a large number of implementations. Our implementations provide insight into how the relationship among the parameters of a machine and the relationship of these parameters to the message sizes can influence performance and thus the choice of the best solution.

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

[4] Z. Bozkus, S. Ranka, G. Fox "Benchmarking the CM-5 Multicomputer," *Proceedings of 4-th Symposium on the Frontiers of Massively Parallel Computation*, pp. 100-107, 1992.

[5] 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.

[6] S.E. Hambrusch, A. Khokhar, "C^3: An Architecture-independent Model for Coarse-Grained Parallel Machines", Technical Report, Purdue University, December 1993.

[7] F. Harary, *Graph Theory*, Addison-Wesley, 1972.

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

[9] F. Thomson Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays . Trees . Hypercubes*, Morgan Kaufmann, 1992.

[10] S. Lillevik, "The Touchstone 30 Gigaflop DELTA Prototype," *Proceedings of 6-th Distributed Memory Computing Conference*, pp. 671-677, 1991.

[11] 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.

[12] 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.

[13] 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.

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