Global Synchronization Algorithms for the Intel iPSC/860

Steven R. Seidel* **Mark A. Davis***

Report Number: RNR-92-027

August 11, 1992

Abstract

In a distributed memory multicomputer that has no global clock, global processor synchronization can only be achieved through software. Global synchronization algorithms are used in tridiagonal systems solvers, CFD codes, sequence comparison algorithms, and sorting algorithms. They are also useful for event simulation, debugging, and for solving mutual exclusion problems. For the Intel iPSC/860 in particular, global synchronization can be used to ensure the most effective use of the communication network for operations such as the shift, where each processor in a one-dimensional array or ring concurrently sends a message to its right (or left) neighbor. Three global synchronization algorithms are considered for the iPSC/860: the `gsync()` primitive provided by Intel, the PICL primitive `sync0()`, and a new recursive doubling synchronization (RDS) algorithm. The performance of these algorithms is compared to the performance predicted by communication models of both the long and forced message protocols. Measurements of the cost of shift operations preceded by global synchronization show that the RDS algorithm always synchronizes the nodes more precisely and costs only slightly more than the other two algorithms.

*Department of Computer Science, Michigan Technological University, Houghton, Michigan 49931-1295

This work was partially supported by NASA Ames Research Center grant NAG 2-757.

1. Introduction

Any node of the Intel iPSC/860 can concurrently send one message and receive one message. For example, when a pair of nodes needs to exchange messages with each other, it is desirable that they exchange those messages concurrently because of the cost savings that concurrency yields. In general, this operation is called an *exchange* and, if the nodes are neighbors, this operation is called a *simple exchange*. Previous work [13] has shown that synchronizing a pair of nodes before an exchange ensures concurrency when the long message protocol is used, and thus reduces communication costs for that operation. (The long message protocol is described further in Section 2.) The cost of synchronization is outweighed by the savings due to concurrency if the messages exchanged are each a few hundred bytes long. This approach has also been used to reduce the cost of certain global communication operations, such as the one-to-all broadcast, the all-to-all broadcast, and the complete exchange, since the best known algorithms for those operations are all based on sequences of exchanges [13].

Many applications, including sequence comparison algorithms, computational fluid dynamics codes, and bin sorting algorithms, configure the processors of the hypercube to form a ring, a one-dimensional array, or a square grid. In these applications each processor performs a computation and then sends a message to its right (or left) neighbor in the ring, array, or row (or column) of the grid. This communication operation is called a *shift*. For example, when a hypercube of n nodes is mapped to a square grid each row of $\sqrt{n}$ nodes will shift a message among the members of that row. When a hypercube is mapped to a ring or one-dimensional array, all n nodes participate in the shift.

Pairwise synchronization is necessary to guarantee concurrency during an exchange of messages using the long message protocol. The same is true for the shift operation, but in this case all nodes of the (sub)cube participating in the shift must be synchronized. The problem of synchronizing all nodes of the hypercube, or one of its subcubes, is called the *global synchronization* problem. Solutions to the global synchronization problem are also useful for solving mutual exclusion problems, logging asynchronous events, assisting in debugging, and generating unique time stamps in distributed databases [5].

Using the forced message protocol for message passing during a shift operation implies that some kind of synchronization is employed to ensure that buffers have been allocated prior to the receipt of incoming messages. (The forced message protocol is described further in Section 2.) One way to do this is to globally synchronize the nodes before a shift that uses the forced message protocol. It is shown that two of the most commonly used synchronization algorithms, `gsync()` and `sync0()`, often do not synchronize the nodes closely enough to guarantee concurrent bidirectional communication. A third algorithm, the recursive doubling synchronization (RDS) algorithm presented here, is shown to synchronize nodes more precisely. This precision ensures concurrency during message passing and reduces the cost of the shift operation by almost a factor of two, compared to its usual cost in practice, for messages of a few kilobytes or more. The cost of the RDS algorithm is in practice only slightly more than the costs of the other two algorithms.

Section 2 provides a brief background on Intel's message passing protocols and their costs. Section 3 describes Intel's `gsync()` algorithm, PICL's `sync0()` algorithm, the RDS algorithm, and the methods used to obtain the timing constants necessary for the RDS algorithm. Section 4 compares the performance of these algorithms in the context of the shift operation.

2. Background

2.1 Message-passing protocols and communication costs

The available message passing protocols and their costs on the iPSC/860 are well documented. (For example, see [2, 13].) The material in this section summarizes those aspects of the message passing system that are relevant to the work presented in the rest of the paper.

Three message passing protocols are available on the iPSC/860. A message of length less than or equal to 100 bytes is sent in a single packet to the destination node. This is called the *short message protocol.* Two protocols are provided for messages longer than 100 bytes. In the *long message protocol* the sending node sends a PROBE message to the destination node. If the receiving node has not already allocated a buffer for the incoming message, it does so. It then sends a REQUEST message to the source node. The PROBE and REQUEST messages are sent using the short message protocol. The source node then sends the message to the destination. The *forced message protocol* is like the long message protocol but skips the PROBE/REQUEST steps. This protocol should only be used when the receiving node has previously allocated a buffer to receive the incoming message.

The *cost* of an operation is described with respect to an ideal global clock. It expresses the amount of time that elapses from when the first node starts the operation to when the last node completes the operation. If more than one node is involved in the operation, as is usually the case, δ denotes the maximum time difference between the times any pair of nodes begins the operation. For example, $\delta = 0$ if the participating nodes all happen to start the operation at exactly the same time.

The cost of a send operation between neighboring nodes using the long message protocol is $am+b_l$, where m is the message length in bytes, a is the cost of transmitting one byte, and b_l is the latency (the l is for *l*ong). (In general, there is also a cost for establishing the circuit that links the sending and receiving nodes. That cost is proportional to the number of links in the circuit. However, the global synchronization algorithms considered here and the shift operation make use of only nearest neighbor communication, so circuit-buildings costs do not need to be included in the cost model described here.) The cost of sending a null (length 0) message using the short message protocol is a constant, denoted by b_s. The cost of a send operation using the forced message protocol is $am+b_s$. Measurements of these constants for the iPSC/860 are given in Table I. The last entry, s, in that table is the time it takes a sending node to return from its call to `csend()` after sending a null message. This constant is discussed further in Section 3.1.

a*	0.36 µsec/byte
b_l	136 µsec
b_s	75 µsec
s	44 µsec

Table I: iPSC/860 communication constants.

* Earlier published estimates of a are as much as 10% higher than the value given here, but this value agrees best with recent measurements.

2.2 Ensuring concurrency during an exchange

According to the communication model described above, when two nodes exchange messages using the long message protocol they must be synchronized to within $\delta < b_s$ μseconds to avoid link contention and ensure concurrency. Measurements have shown that slightly more precise synchronization is actually required (65 μseconds [4]) but to simplify the analysis that follows $b_s = 75$ μseconds will be used as the bound. When it is critical to the analysis, the more precise 65 μsecond bound will be noted.

As shown in Figure 2-1, when $\delta < b_s$ the cost of the simple exchange is $am+b_l+\delta$. (The contribution of δ to the cost is not shown explicitly but is easily inferred in Figure 2-1 and those that follow. The costs illustrated in this series of figures are the minimums obtainable in each case.) In practice, however, the nodes are not this closely synchronized. When $b_s \le \delta \le am+b_l/2$ the simple exchange of long messages proceeds as shown in Figure 2-2. In this case link contention causes messages to be passed sequentially because each node can send at most one message and receive at most one message at the same time. The cost in this case is $2am+(3/2)b_l$. Note that this cost does not involve δ. Finally, when $\delta > am+b_l/2$ the PROBE from the righthand node in Figure 2-2 is not entirely concurrent with the long message from the lefthand node and the cost becomes $2am+(3/2)b_l+\delta'$, where $\delta' = \delta - (am+b_l/2)$.

To ensure concurrent bidirectional communication during the simple exchange each node should send a null (zero length) message to the other and wait for a reply before sending the long message. This version of the simple exchange is illustrated in Figure 2-3 for the case of $\delta = b_s$. The cost depends upon the amount of overlap, if any, of the two null messages. In general, the cost is $am+b_l+b_s+\delta$. It is easy to show that this approach to synchronizing the simple exchange costs less than the nonsynchronized version when $b_s \le \delta \le am+(1/2)b_l-b_s$. Since $am+(1/2)b_l-b_s \approx am$, it is convenient to simplify this range to $b_s \le \delta \le am$. In practice it is reasonable to assume that δ falls within this range because b_s is small and, for messages of a few kilobytes or more, am is relatively large. It is therefore safe to say that in practice the cost of a simple exchange of messages can be reduced by almost half if the nodes are first synchronized by an exchange of null messages.

Figure	operation	protocol	cost	δ
	send	short	b_s	
	send	long	$am + b_l$	
	send	forced	$am + b_s$	
2-1	simple exchange	long	$am + b_l+\delta$	$\delta < b_s$
2-2	simple exchange	long	$2am + (3/2)b_l$	$b_s \le \delta \le am$
2-3	synch'd simp. exch.	long	$am + b_l+b_s+\delta$	$\delta \ge 0$
2-4	synch'd simp. exch.	forced	$am + 2b_s+\delta$	$\delta \ge 0$
2-5	shift	long	$am + b_l+\delta$	$\delta < b_s$
2-6	shift	long	$2am + (3/2)b_l$	$b_s \le \delta \le am$
2-7	shift	forced	$am + b_s+\delta$	$\delta \ge 0$

Table II: Costs of message passing operations.

The forced message protocol assumes that the destination node has a buffer into which to put the incoming message. This is accomplished by having the destination node allocate a buffer by a call to `irecv()` prior to the message transmission. One way to notify the sending nodes that these buffers have been allocated is to exchange null messages, as shown in Figure 2-4. This amounts to synchronizing the simple exchange, just as when the long message protocol was used.

Again, the cost depends upon the amount of overlap of the two null messages. In this case the cost is $am + 2b_s + \delta$, for all $\delta \geq 0$.

Table II contains a summary of the costs given above. Also included in that table are cost estimates of the shift operations described in the next section.

2.3 Ensuring concurrency during a shift

Event simulation, tridiagonal matrix solvers [7], computational fluid dynamics codes [11], sequence comparison algorithms [8, 9], and some bin sorting algorithms [14] all use a communication operation called a *shift*. In a shift, processors in a hypercube that are mapped to a one-dimensional array or ring each send a message to their right (or left) neighbor. The cost analysis of the shift operation is analogous to that of the simple exchange.

When the long message protocol is used and $\delta < b_s$ this operation has a cost of $am+b_l+\delta$, as shown in Figure 2-5. (In this, and the following figures, the latency b_l is assigned to the PROBE and REQUEST steps of the long message protocol and the remaining cost am is assigned to the transmission of the long message itself. The actual costs of each step in the protocol have not been individually measured.) When $b_s \leq \delta \leq am$ the cost increases to $2am+(3/2)b_l$, as shown in Figure 2-6. Again, this increased cost is due to the fact that each node is limited to sending one and receiving one message at a time. To ensure concurrency when using the long message protocol, the nodes in the array or ring should be globally synchronized prior to the shift.

The cost of a shift using the forced message protocol is $am+b_s+\delta$, for all $\delta \geq 0$, as shown in Figure 2-7. Using the forced protocol implies that some form of synchronization has been used to ensure that buffers have been allocated prior to the receipt of the incoming messages. One way to do this is to globally synchronize the nodes. Thus, global synchronization can be useful even when shifting using the forced message protocol.

2.4 Message buffer management

In general, care must be taken in order to get the best performance from the communication network. The cost of communication operations can be predicted with the greatest reliability when the communication buffer system is in a known state. Unless those buffers are managed carefully, random message traffic can occur on the network and result in poor performance. Also, receiving unexpected messages can lead to imbalances in the number of message buffers in the nodes which directly leads to an increase in system message traffic on the network.

The operating system of the nodes has a small set of buffers reserved to hold headers for messages. The forward flow control management of these buffers allows several messages to be sent from one node to another before a response is received. Once the buffer pool in a receiving node is exhausted, that node will refuse to accept additional incoming messages until it has indicated to the sender that it has one or more buffers available for the receipt of new messages. In this situation the buffer management software in each node intervenes by spontaneously sending a message to release available buffers [12]. Such message traffic is difficult to predict and can adversely affect measurements of the message passing system and the performance of applications. Applications that continually balance the number of messages sent to and received from a particular node avoid this problem since the headers of the application messages are also used to transmit the necessary buffer information. In particular, the short message protocol and the PROBE message of the long message protocol consume flow control buffers. The forced message protocol bypasses this management system since it is assumed that the destination node has allocated a buffer prior to receipt of the message. In actual applications, balanced message traffic must be maintained in order to ensure the most effective use of the network.

When timing a communication operation, it is usually repeated many times and the cumulative cost is averaged. In these situations there is usually no complementary (backward) flow of messages. The program used to determine the constant s is one such example. Accurate measurements of this constant could only be obtained by balancing the flow of messages. In the measurements given in Sections 3 and 4, care was taken to balance the message traffic so that spontaneous flow control messages did not need to be sent by the node operating systems.

3. Synchronization Algorithms

Three algorithms for global synchronization are considered. The first, called `gsync()`, is provided as a primitive operation in NX/2, the node operating system of the iPSC/860. The second, called `sync0()`, is provided with the Portable Instrumented Communication Library [6]. The third is called the recursive doubling synchronization (RDS) algorithm.

3.1 `gsync()`

The `gsync()` function is provided as a communication primitive to globally synchronize the nodes of the iPSC/860* [10]. This algorithm is also called the butterfly barrier and was originally proposed for shared memory machines [1, 3]. Let d denote the dimension of the hypercube. In the `gsync()` algorithm each node sends a null message to each of its d neighbors, starting with the neighbor whose most significant bit differs from its own. It waits for a reply to each message before sending the next.

The cost of `gsync()` depends on how closely synchronized the nodes are when each makes its call, that is, the cost depends on δ. If $\delta < b_s$, then the cost of `gsync()` is $db_s+\delta$, as shown in Figure 3-1. While that figure indicates that each simple exchange is done concurrently, it is assumed that some nodes lag the others by up to δ time units and so some of the exchanges are only partially overlapped in time. The slower nodes and all of their neighbors thus finish as much as δ time units later than the remaining nodes. Let the maximum time difference, relative to a global clock, between the times any pair of nodes completes their calls to a synchronization function be called the *precision* of that function. In this case the precision of `gsync()` is no greater than δ.

In practice it is unlikely that all nodes are synchronized to within b_s μseconds when they reach their calls to a synchronization function. (If it happens that $\delta < b_s$, then global synchronization is not required to achieve concurrency during message passing.) It is now shown that if $\delta \geq b_s$, then the cost of `gsync()` can be considerably higher than $db_s+\delta$ and its precision depends on both δ and d. Assume that node 7 is delayed by $\delta = b_s$ μseconds, as shown in Figure 3-2. Because of this delay, node 7 does not immediately send its message to node 3. In the worst case, shown in the figure, that message can be blocked during steps 2 and 3. (The dashed arrow shows that receive port contention has blocked the message from node 7.) When the receive port on node 3 is finally available, node 7 can then send its message to node 3 and to the remainder of its neighbors. This amounts to a recursive doubling broadcast rooted at node 7, as shown in the last three steps of Figure 3-2. Note that the choice of node 7 in this example was arbitrary.

Let s be the time that it takes the source node to complete a send. Since a source node finishes sending before the destination node finishes receiving, $s < b_s$. Measurements indicate that $s = 44$ μseconds (see Table I). Figure 3-3 is a timing diagram of Figure 3-2. That diagram shows that node 7 will be out of sync with node 0 by $3(b_s-s)$ μseconds when node 7 returns from

*The implementation of `gsync()` is described here with permission of Intel.

its call to `gsync()` and, in fact, no node is synchronized with node 7. In general, the precision of `gsync()` is $d(b_s-s)$ μseconds in a hypercube of dimension d when $\delta \geq b_s$. The total cost of `gsync()` in this case is $2db_s$, about twice the cost when $\delta < b_s$.

A complete analysis of the cost and precision of `gsync()` is beyond the scope of this paper. The examples given above allow us to conclude that when $\delta < b_s$ the cost of `gsync()` is $db_s+\delta$ and its precision is δ. Also, when all but one of the nodes is initially synchronized with the others and when $b_s \leq \delta \leq db_s$ the cost is approximately $2db_s$, and when $\delta > db_s$ the cost is $db_s+\delta$. In these two cases the precision of `gsync()` is $d(b_s-s)$ because the final steps of the algorithm constitute a recursive doubling broadcast rooted at the "slow" node.

3.2 `sync0()`

The `sync0()` synchronization algorithm is provided with the Portable Instrumented Communication Library [6]. If it happens that $\delta < b_s$, `sync0()` operates as shown in Figure 3-4. In the first step each node whose high order bit is 1 sends a message to its neighbor whose high order bit is 0. The subcube of $d/2$ nodes whose high order bits are 0 is then synchronized using the same sequence of simple exchanges that are used in `gsync()` (except the order in which the sends are done is reversed). On the last step each node in that subcube sends a message to its neighbor whose high order bit is 1. The cost of `sync0()` in this case is $(d+1)b_s+\delta$ and its precision is δ.

Figure 3-5 illustrates the operation of `sync0()` when node 7 is delayed by $\delta = b_s$ μseconds. In this case the first message that node 7 tries to send can be blocked at steps 2 and 3. The timing diagram in Figure 3-6 shows that when the algorithm terminates, nodes 3 and 4 will be out of sync by $d(b_s-s)$ μseconds. The cost of `sync0()` in this case is $(2d+1)b_s$. A complete analysis of the cost and precision of `sync0()` is also beyond the scope of this paper. Such an analysis is more complex than in the case of `gsync()` because the `sync0()` algorithm is different for nodes in the high and low halves of the cube. It remains that `sync0()` uses an algorithm essentially identical to `gsync()` on a subcube of dimension $d-1$, so it can be expected to have about the same cost and precision as `gsync()`.

3.3 The RDS algorithm

The third synchronization algorithm considered here is the *recursive doubling synchronization* (RDS) algorithm. This global synchronization algorithm has three phases. The first phase is a reverse of a recursive doubling broadcast, as shown in Figure 3-7. This serves to create a barrier that brings all of the nodes into a known state. This is followed by a recursive doubling broadcast rooted at node 0. The nodes do not complete this phase at the same time, but it is easy to determine the relative time at which each node does complete this phase. Based on that determination, in the third phase each node does a busy wait until all messages have propagated to their destinations. This waiting phase, with waiting times based on measurements of message passing costs, ensures that all nodes are synchronized at the end of the third phase.

The timing diagram for the RDS algorithm is shown in Figure 3-8. Assuming node 0 is the root, the last node to receive a message in phase 2 is node 2^d-1. Each node i then waits for w_i μseconds after it has finished the phase 2 broadcast to become synchronized with node 2^d-1. The time required for the message broadcast from the root to propagate to each node i in the recursive doubling spanning tree is used to determine w_i. Let $|i|_1$ denote the number of 1 bits in the node number of node i. The recursive doubling spanning tree is chosen so that $|i|_1$ is the level of node i in that tree. Each node must wait some integer multiple of (b_s-s) time units in order to be synchronized with node 2^d-1. That multiple for node i is just d minus the level of node i in the tree, or $d - |i|_1$. Thus, $w_i = (d - |i|_1)(b_s-s)$. (If only a subcube needs to be synchronized, then only

those bits which differ in the subcube are counted in $|i|_1$.) Each node i will perform a busy wait by executing an empty `while` loop.

The cost of the RDS algorithm is easy to determine and does not depend on the relation of δ and b_s. The cost of phase 1 is just the time it takes the "slowest" node's message to reach node 0, namely, $db_s+\delta$. The cost of phase 2 is the cost of a recursive doubling broadcast, db_s. The waiting time of phase 3 does not contribute to the cost of the algorithm because the busy waiting occurs only while other nodes are waiting to receive synchronization messages during phase 2. Thus, the total cost of the RDS algorithm is $2db_s+\delta$. This is comparable to the costs of `gsync()` and `sync0()` except in the extreme case when $\delta < b_s$. The precision of the RDS algorithm is 0, for all $\delta \geq 0$.

To obtain this precision in practice requires accurate estimates for s and b_s. Recall that the parameter s is the time it takes the sending node to complete its send of a null message using the short message protocol, and b_s is the time that the destination node finishes receiving the null message relative to when the sending node started. The latter is just the cost (latency) of the short message protocol. The value of s is easily measured since it depends only on the performance of a single node. The estimate used here is $s = 44$ μseconds. The value of b_s was measured by timing the (synchronous) send/reply operation and dividing by 2. The estimate used here is $b_s = 75$ μseconds. These estimates are each based on 10,000 trials obtained using the Intel `dclock()` system primitive.

An empty `while` loop was timed in order to implement the busy wait in the third phase of the RDS algorithm. Care was taken to ensure that compiler optimizations did not eliminate the explicit execution of such loops. Figure 3-9 shows that interrupts periodically add about 50 μseconds to the cost of a fixed number of iterations. Since this variation is within the 65 μsecond tolerance required for node synchronization mentioned earlier, and since disabling the interrupts can have an adverse effect on other users, this variation was ignored. (On the other hand, analogous work on the iPSC/2 necessitated disabling the interrupts because their magnitude was larger, probably due to slower node processors, while the synchronization tolerance was the same 65 μseconds due to the similarity of interconnection hardware used on the iPSC/2 and iPSC/860. See [4] for details about the implementation of the RDS algorithm on the iPSC/2.) The costs of various numbers of iterations of an empty `while` loop are given in Figure 3-10. Ignoring the outlying data points, a line fit to those data gives the cost of k iterations to be $w_i = 0.126k+2.0$ μseconds. It follows that a delay of w_i μseconds can be obtained with an empty `while` loop of $k = (w_i-2.0)/0.126$ iterations.

3.4 The effect of system interrupts

It is now shown that when a call to `gsync()` precedes a shift operation, synchronization messages can sometimes contend for communication links with shifted messages because of the imprecision of `gsync()` and the random occurrence of system interrupts. Even when the forced message protocol is used in the shift, the actual cost of the shift can be twice the predicted cost of $am+b_s$. `sync0()` is also subject to this insecurity because it uses essentially the same algorithm as `gsync()` on a subcube of dimension $d-1$. The discussion that follows thus considers only the behavior of `gsync()`.

Suppose that a shift is preceded by a call to `gsync()` in a hypercube of 4 nodes and that the shift is around a ring embedded by the usual binary reflected Gray code. Also assume that node 0 is initially out of sync by $\delta > b_s$ μseconds so that it acts as the root in a recursive doubling broadcast during the latter part of the `gsync()` algorithm, as described in Section 3.1. In the absence of system interrupts, the broadcast and subsequent shift proceed as shown in Figure 3-11(a). Note that the cost of shifting messages of a few kilobytes or more is much greater than the cost of `gsync()`.

Now suppose that a system interrupt occurs in node 2 just after it has received the first message broadcast from node 0. This delays node 2 for approximately 50 μseconds, an amount that is slightly greater than s. Because of this delay, node 1 can begin its shift to node 3 before node 2 can send its synchronization message to node 3. When node 2 returns from its interrupt it must wait for node 1 to complete its shift. Thus, the shifts from nodes 1 to 3 and from nodes 3 to 2 occur sequentially rather than concurrently. This situation is illustrated in Figure 3-11(b). In this case the cost of the shift operation is about twice what it was when no interrupts occurred.

While such occurrences of interrupts during synchronization have not been observed directly, there are two factors that contribute to the likelihood that interrupts will interfere with `gsync()` and `sync0()`, thus increasing the cost of the shift operation. First, the number of interrupts that occurs grows with the size of the hypercube simply because there are more nodes liable to interruption. Second, in the absence of interrupts each pair of neighboring nodes in the ring finishes the recursive doubling broadcast within approximately s μseconds of each other. This is a consequence of the structure of the broadcast tree and the Gray code embedding of the ring. The timing relationship among nodes 1, 3, and 2 depicted in Figure 3-11(a) thus occurs at many sites in larger hypercubes. One would therefore expect to see fewer concurrent shifts as the hypercube grows. This is exactly what was observed in the performance measurements given in Section 4.

Recall that when $\delta \geq b_s$, the precision of `gsync()` is $d(b_s - s)$ μseconds. However, with respect to the Gray code embedding, each pair of neighboring nodes in the ring is out of sync by only $b_s - s$ μseconds because they are on adjacent levels of the recursive doubling broadcast tree. In the absence of system interrupts `gsync()` would thus be expected to successfully ensure concurrency during the shift operation. In the presence of interrupts `gsync()` is less likely to be successful, for the reasons described above. The low success rate of `gsync()` and `sync0()` presented in Section 4 for the iPSC/860 is therefore attributed to the affect of system interrupts.

Interrupts from the node operating system can also affect the precision of the RDS algorithm, but not so much that they increase the cost of the shift operation. It can be deduced from Figure 3-9 that each node services an interrupt about every 10 milliseconds. Interrupts that occur in nodes during phase 1 of the RDS algorithm add only to the cost of synchronization and do not affect the precision of the algorithm. Because interrupts occur with a definite period, each node services the same number of interrupts (± 1) as any other node during the last two phases of the RDS algorithm, so no node is delayed more than 50 μseconds by interrupts with respect to any other node. Since the tolerance for synchronization is 65 μseconds, a pair of neighboring nodes in the ring that are out of sync by 50 μseconds should not affect the cost of the shift operation. The busy waits during phase 3 prevent system interrupts from interfering with synchronization messages and shift messages that can occur in `gsync()` and `sync0()`. Figure 3-12 is the analog of Figure 3-11(b) for the RDS algorithm. The busy wait by node 1 prevents node 1 from beginning to shift its message to node 3 even though node 2 has been temporarily delayed by an interrupt.

4. Observed Performance

There is no global clock available to the nodes of the iPSC/860 so it is not possible to directly measure how closely the nodes are synchronized. This section presents indirect evidence of synchronization precision based on the observed costs of the shift operation. Measurements were made of the time it takes to synchronize and then shift a message one hop around a ring of nodes. The `gsync()`, `sync0()`, and the RDS algorithms were interchanged, and message lengths, protocols and ring sizes were varied.

4.1 Observed shifting costs

The data collected from repeatedly synchronizing and shifting various sizes of messages in rings using the long and forced protocols and `gsync()`, `sync0()`, and the RDS algorithms can be examined in several ways. Figures 4-1 through 4-3 show the observed and predicted times for the `gsync()` and RDS algorithms used before shifts using the long and forced message protocols for d=2, 5, and 7 for the iPSC/860. Only the times for the shift operation are plotted since the synchronization cost depends on how closely synchronized the nodes are from one pass through the loop to the next. The minimum predicted cost of a shift using the forced message protocol, $am+b_s$, is included in these graphs as a reference. Each observed time plotted in these figures is the average of the longest time recorded in each trial. Not as much data could be obtained in the case of `sync0()`, but those graphs include what was available.

When the forced message protocol is used to shift messages after synchronization by the RDS algorithm the observed performance agrees well with the predicted cost and is considerably less than the cost of shifting after `gsync()` or `sync0()`. For sufficiently large messages the RDS algorithm reduces the cost of the shift operation by several milliseconds compared to those algorithms. This difference is attributed to the affect of system interrupts. The savings increase with the size of the ring.

A similar relationship holds between the performance of these algorithms when the long message protocol is used for shifting. In this case, however, none of the algorithms are able to consistently reach the minimum cost, although the RDS algorithm always performs better than the other two. The poor performance of all three algorithms when shifting using the long message protocol is attributed to the forward flow control of message buffers imposed by the node operating system. The occurrence of interrupts is the most likely reason why `gsync()` and `sync0()` perform less well than the RDS algorithm.

Figure 4-4 shows how the various protocols and synchronization algorithms perform for various size hypercubes with the length of the shifted message held constant at 32K bytes. These data also indicate that consistent performance is obtained only when shifting is done using the RDS algorithm and the forced message protocol.

4.2 Frequency with which concurrency is obtained

A second way of examining how well the three algorithms synchronize the nodes is to consider how frequently concurrent shifting is achieved. A *success* is defined to be a trial where every node in the ring concurrently sends and receives the shifted messages. A *failure* occurs if one or more of the nodes does not send and receive the messages concurrently in a single trial. Successes and failures can be identified by their proximity to the predicted cost in each case. Since these costs differ by almost a factor of 2, successes and failures are easy to distinguish. In the data presented here, a failure was defined to be a shift operation whose cost was 1 millisecond greater than the predicted minimum cost. Otherwise the trial was a success. This is a conservative definition of failure because it is less than the difference in predicted costs between concurrent and nonconcurrent shifts for the range of message lengths (4K to 64K bytes) considered here. In particular, a concurrent shift of a message of 4K bytes using the forced protocol is predicted to cost 0.36×4096+75 = 1.55 milliseconds, while a nonconcurrent shift costs twice that amount. If there is no concurrency during the shift the measured cost will always exceed 2.55 milliseconds and will be regarded as a failure. While the difference between the costs of concurrent and nonconcurrent shifts becomes more pronounced as the message length grows, the 1 millisecond threshold was held constant.

The percentage of successes out of 50 trials is plotted in Figures 4-5 to 4-7. The small number of trials was chosen because the long message protocol performs poorly for large

numbers of trials. This is caused by the message buffer imbalance created by repeatedly performing the same operation. These data show that the RDS algorithm achieves a consistently high success rate when messages are shifted using the forced message protocol. Note that the success rates of `gsync()` and `sync0()` fall off as the size of the hypercube increases, as anticipated in Section 3.4. On the other hand, the frequency of successes is erratic and drops off significantly when the long message protocol is used with the RDS algorithm. However, in all cases the success rate of the RDS algorithm is greater than those of `gsync()` and `sync0()`.

4.3 Minimum message sizes

The performance results given above show that globally synchronizing the nodes using the RDS algorithm before a shift using the long message protocol can reduce the cost of the shift compared to the use of other synchronization algorithms. Since synchronization adds to the cost of the shift, it must be determined when it will be economical to synchronize before the shift. The minimum message size for which it should be advantageous to use the RDS algorithm before a shift using the long message protocol can be determined by solving for the message length m in the inequality below. The left hand side is the cost of sending a message using the long message protocol with no synchronization. The right hand side is the cost of a concurrent shift using the long message protocol plus the cost of the RDS algorithm. The term δ is included on the left hand side because it determines when the "slowest" node will complete the shift. That term also appears on the right hand side because, while it no longer contributes to the cost of the shift, it is part of the cost of the RDS algorithm.

$$2am + \frac{3}{2}b_l + \delta \;\geq\; am + b_l + 2db_s + \delta,$$

$$\text{so } m \;\geq\; \frac{(2db_s - b_l/2)}{a}.$$

Using the constants a, b_s, and b_l from Table I, the minimum message sizes for which synchronization should be used are given in Table III. No empirical confirmation of these minimum message lengths is offered here. That is left to observations of the actual applications in which shifting is used.

d=2	d=3	d=4	d=5	d=6	d=7
645	1062	1478	1895	2312	2728

Table III: Minimum message lengths (in bytes) for which RDS synchronization is recommended when using the long message protocol.

When the forced message protocol is used for the shift, global synchronization can be used to ensure that all necessary message buffers have been allocated. The analysis in Section 3.4 illustrated one case when `gsync()` and `sync0()` do not reliably synchronize the nodes and in turn cause the cost of the shift to be about twice the minimum. It follows that if global synchronization is used before a shift it is always preferable to use the RDS algorithm, regardless of the length of the messages shifted and the size of the hypercube. This reasoning is confirmed by the performance data given earlier in this section for message lengths ranging from 4K to 64K bytes and for hypercubes of all sizes.

5. Summary

Three global synchronization algorithms for the Intel iPSC/860 were described. The costs of these algorithms are comparable in practice, so the choice of algorithm can be based on the precision with which these algorithms synchronize the nodes. This precision was measured indirectly through observations of the cost of the shift operation. That operation has the property that its observed cost tends to differ by a factor of two depending upon whether or not concurrent bidirectional communication is achieved. Both the long and forced message protocols were used to shift messages after the nodes were synchronized. Also described was how the random occurrence of system interrupts can adversely effect the cost of shifting after a call to `gsync()`.

The RDS algorithm combined with the shift using the forced message protocol ensures concurrent shifts greater than 99% of the time for hypercubes of all sizes and for messages of at least 4K bytes. In a ring of only four nodes the success rate of `gsync()` is similar to that of the RDS algorithm when the forced message protocol is used. As the size of the hypercube grows it becomes more likely that an interrupt will cause synchronization and shift messages to contend during `gsync()`, but the RDS algorithm is not subject to this problem in practice. The data confirm that `gsync()` loses its effectiveness while the RDS algorithm continues to reliably synchronize the nodes as the hypercube grows. For example, the frequency with which `gsync()` achieves concurrency ranges from about 90% for d=2 to less than 5% for d=7. Based on the data available, `sync0()` performs at a level intermediate to `gsync()` and the RDS algorithm.

The RDS algorithm does not perform as consistently when messages are shifted using the long message protocol. This inconsistency is attributed to the forward flow control protocol used for message buffer management by the node operating system. However, the data show that the RDS algorithm is still preferable to `gsync()` and `sync0()` and the cost of the RDS algorithm is fully amortized when the messages being shifted are several kilobytes long. The actual crossover point depends on the size of the hypercube. That point varies from about 650 bytes for d=2, to 2730 bytes for d=7.

References

1. T. Axelrod, Effects of synchronization barriers on multiprocessor performance, *Parallel Computing 3* (1986), 129-140.
2. S. Bokhari, Communication overhead on the Intel iPSC/860 hypercube, ICASE Interim Report 10, ICASE, May 1990.
3. E. D. Brooks III, A multitasking kernel for the C and FORTRAN programming languages, UCID-20167, Lawrence Livermore National Laboratory, 1984.
4. M. A. Davis and S. R. Seidel, Global synchronization algorithms for the Intel iPSC/860 and iPSC/2 hypercubes, Tech. Rep. 92-04, Dept. of Computer Science, Michigan Tech. Univ., May 1992.
5. J. C. French, A global time reference for hypercube multicomputers, IPC-Tech. Rep.-88-10, School of Engineering and Applied Science, Univ. of Virginia, October 1988.
6. G. A. Geist, M. T. Heath, B. W. Peyton and P. H. Worley, PICL: A Portable Instrumented Communication Library, C reference manual, ONL/TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July, 1990.
7. M. T. Heath and C. H. Romine, Parallel solution of triangular systems on distributed-memory multiprocessors, *SIAM Journal of Scientific Statistical Computing. 9, 3* (May 1988), 558-588.
8. X. Huang, W. Miller, S. Schwartz and R. Hardison, Parallelization of a local similarity algorithm, *Computer Applications in the BioSciences 8* (1992), 155-165.
9. X. Huang, A space-efficient parallel sequence comparison algorithm for a message-passing multiprocessor, *International Journal of Parallel Programming 18, 3* (June 1989), 223 - 239.
10. *iPSC/2 and iPSC/860 Release 3.3 Software Product Release Notes*, Intel Scientific Computers, June 1991.
11. A. Jameson, W. Schmidt and E. Turkel, Numerical solution of the Euler equations by finite volume methods using Runge-Kutta time-stepping schemes, AIAA Paper 81-1259, June 1981.
12. P. Pierce, Intel SSD, *Personal communication*, April 1991.
13. S. R. Seidel, M. Lee and S. Fotedar, Concurrent bidirectional communication on the Intel iPSC/860 and iPSC/2, *Proc. of the Sixth Distributed Memory Computing Conf.*, April 1991, 283-286.
14. Y. Won and S. Sahni, A balanced bin sort for hypercube multicomputers, *Journal of Supercomputing 2* (1988), 435-448.

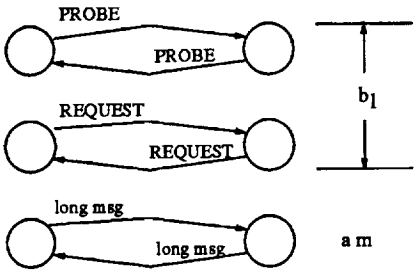

Fig. 2-1: Simple exchange of long messages, $\delta < b_s$.

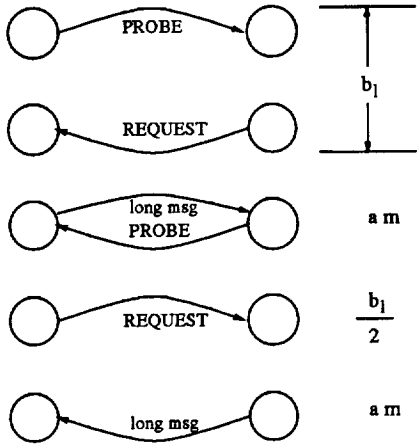

Fig. 2-2: Simple exchange of long messages, $b_s \leq \delta \leq am$.

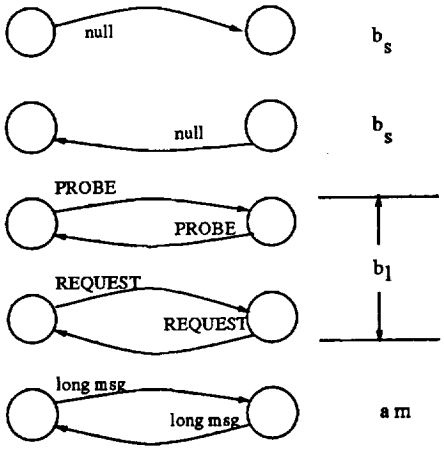

Fig. 2-3: Synchronized simple exchange, long protocol.

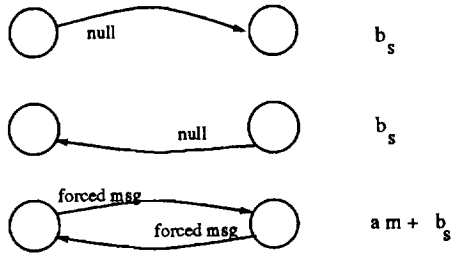

Fig. 2-4: Synchronized simple exchange, forced protocol.

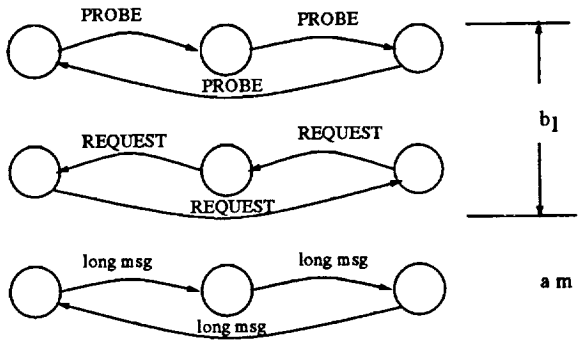

Fig. 2-5: Concurrent shift, long protocol, $\delta < b_s$.

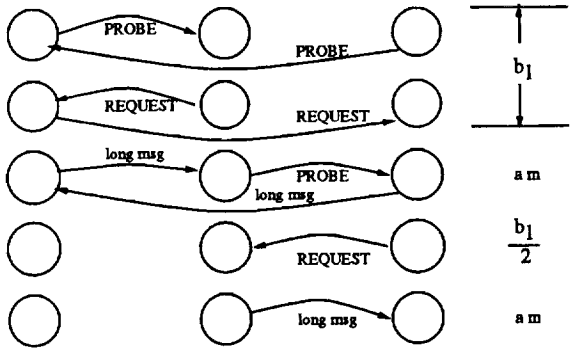

Fig. 2-6: Non-concurrent shift, long protocol, $b_s \leq \delta \leq am$.

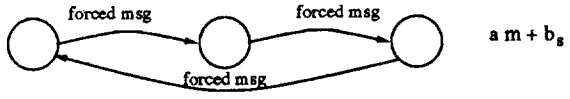

Fig. 2-7: Concurrent shift, forced protocol.

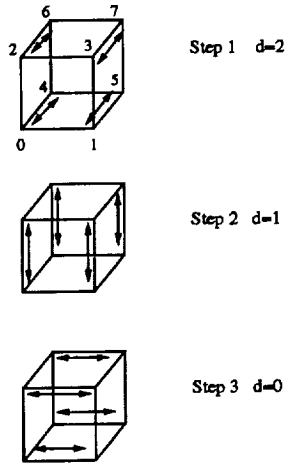

Fig. 3-1: *gsync()* with initally synchronized nodes.

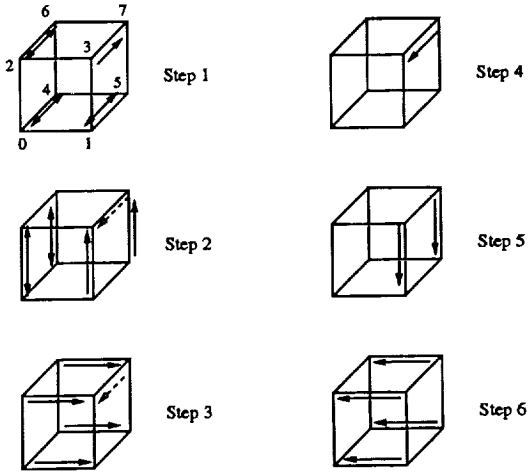

Fig. 3-2: *gsync()* with node 7 delayed b_s μsec.

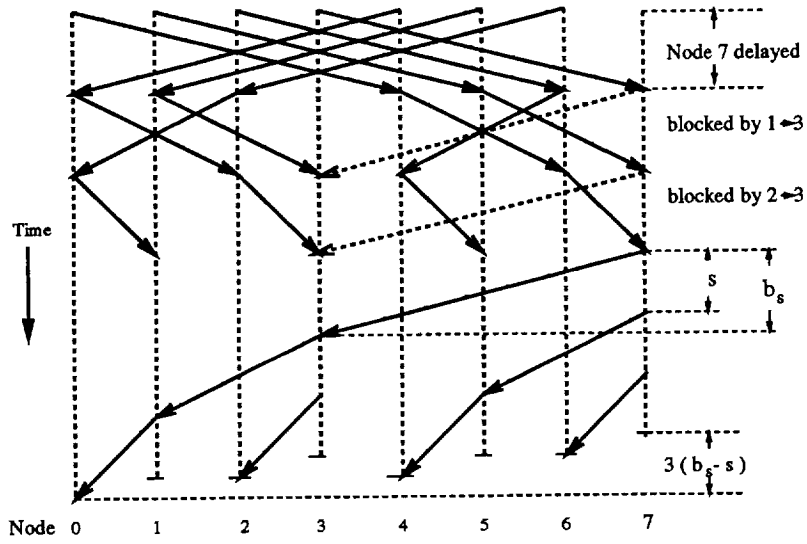

Fig. 3-3: *gsync()* timing diagram (node 7 delayed).

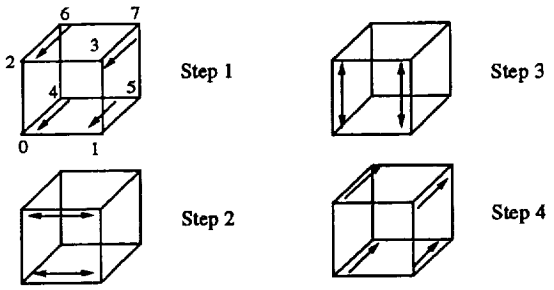

Fig. 3-4: *sync0()* with initally synchronized nodes.

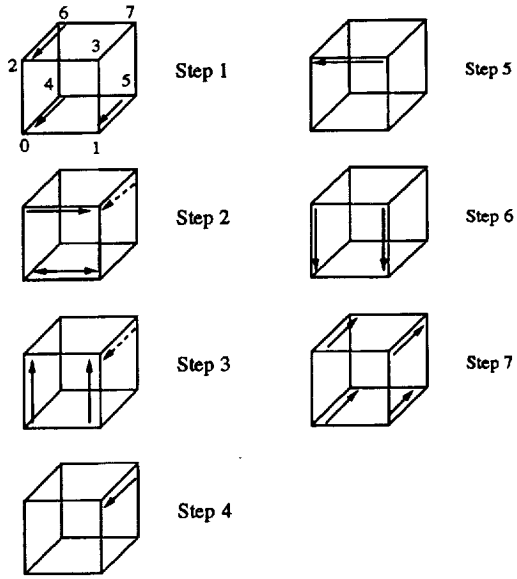

Fig. 3-5: *sync0()* with node 7 delayed.

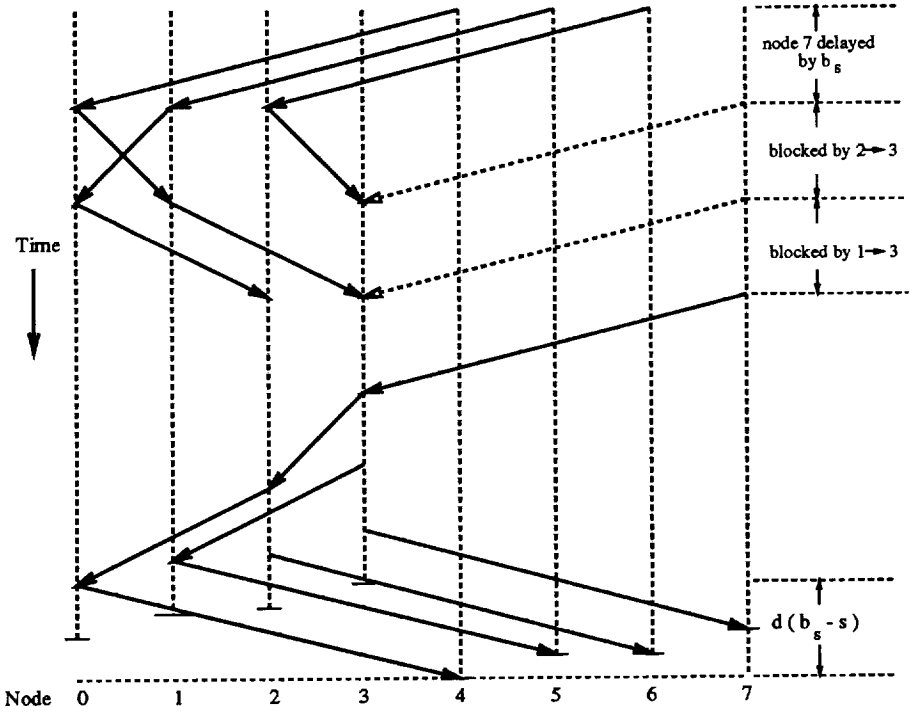

Fig. 3-6: *sync0()* timing diagram (node 7 delayed).

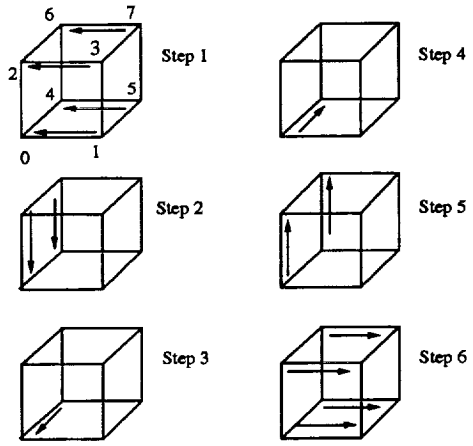

Fig. 3-7: Recursive doubling synchronization (RDS) algorithm.

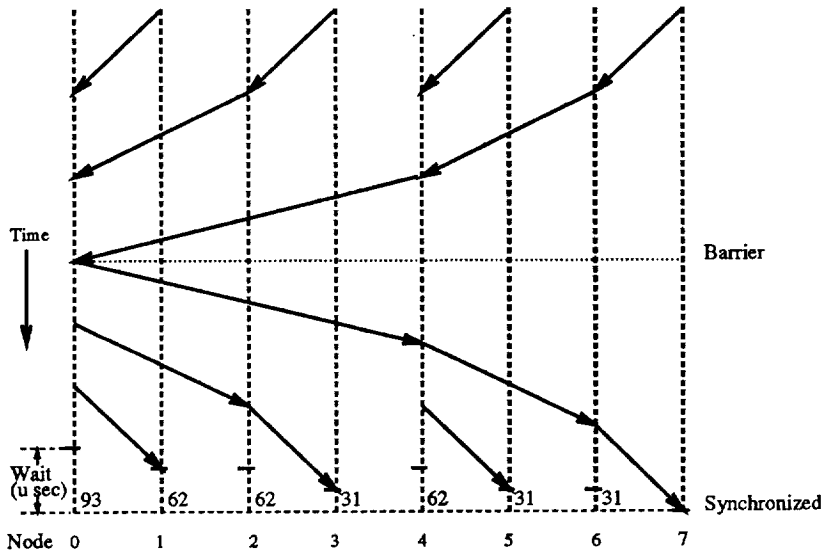

Fig. 3-8: RDS algorithm timing diagram.

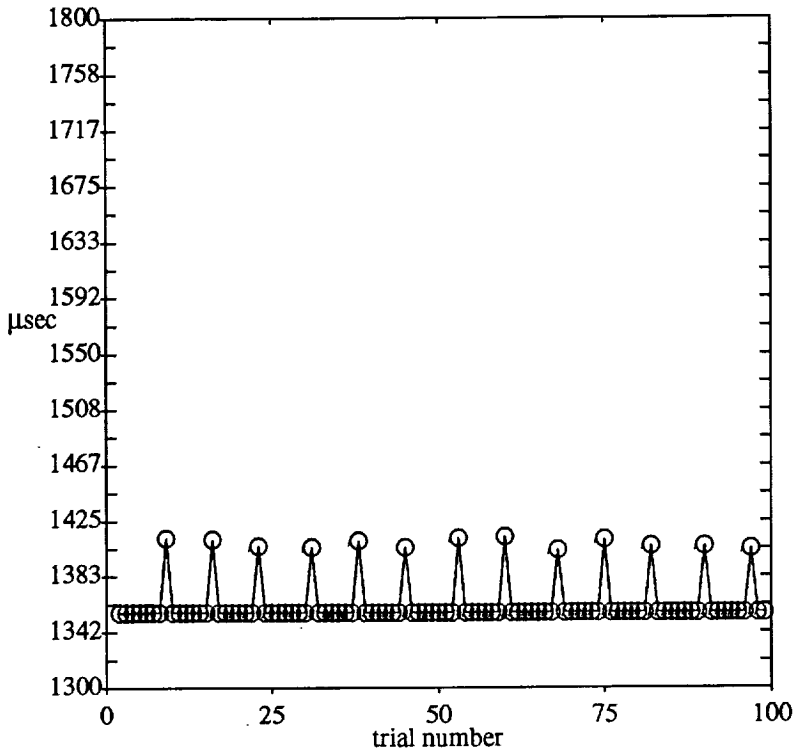

Fig. 3-9: Empty while loop timed by the dclock() on the iPSC/860, each trial is 10833 iterations.

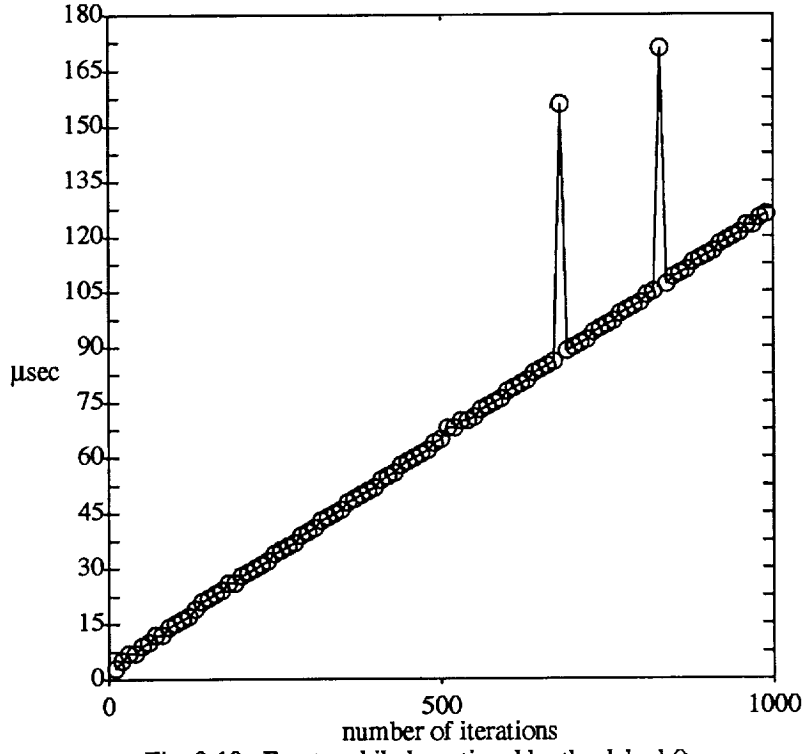

Fig. 3-10: Empty while loop timed by the dclock() on the iPSC/860.

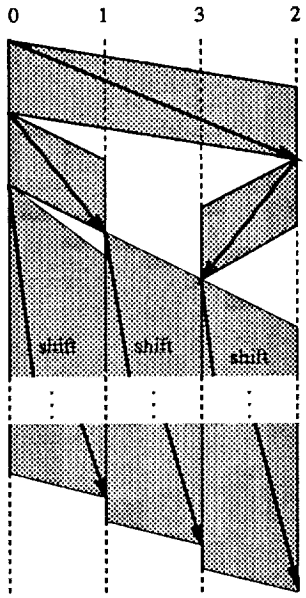

Fig. 3-11(a): *gsync()* with no interrupts.

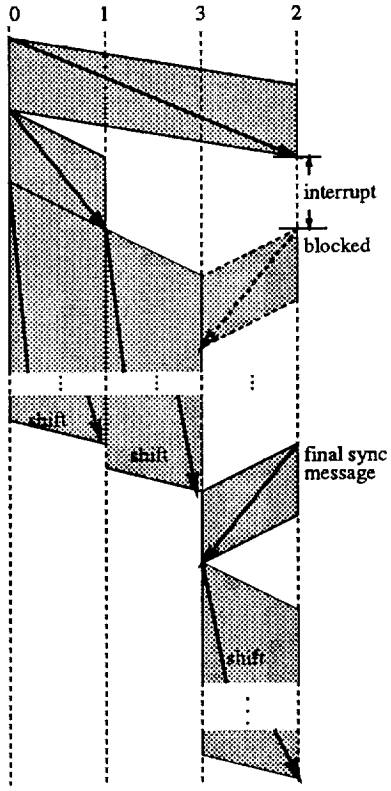

Fig. 3-11(b): The affect of a system interrupt on *gsync()*.

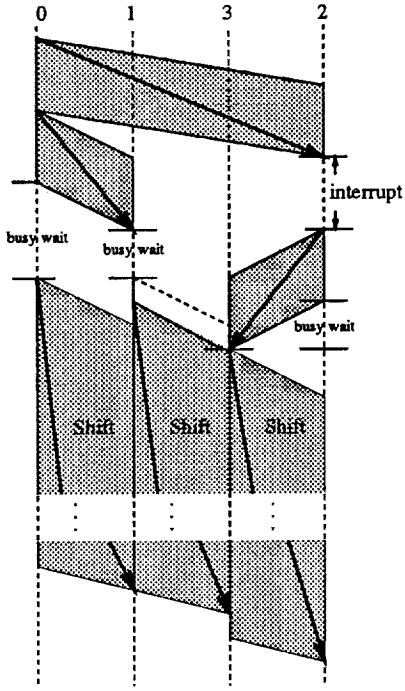

Fig. 3-12: The affect of an interrupt on RDS.

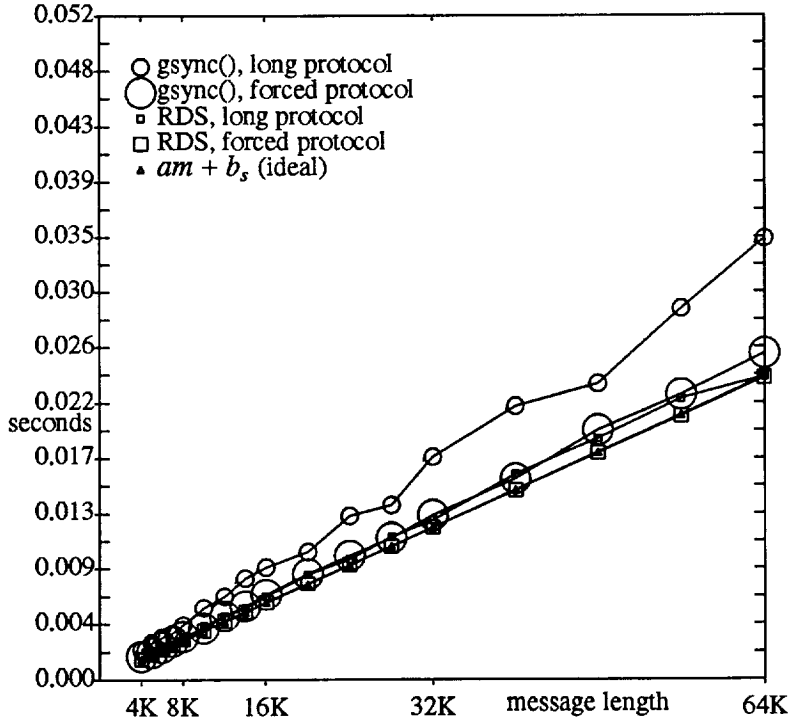

Fig. 4-1: Cost of shifts in a 4-node ring on the iPSC/860.

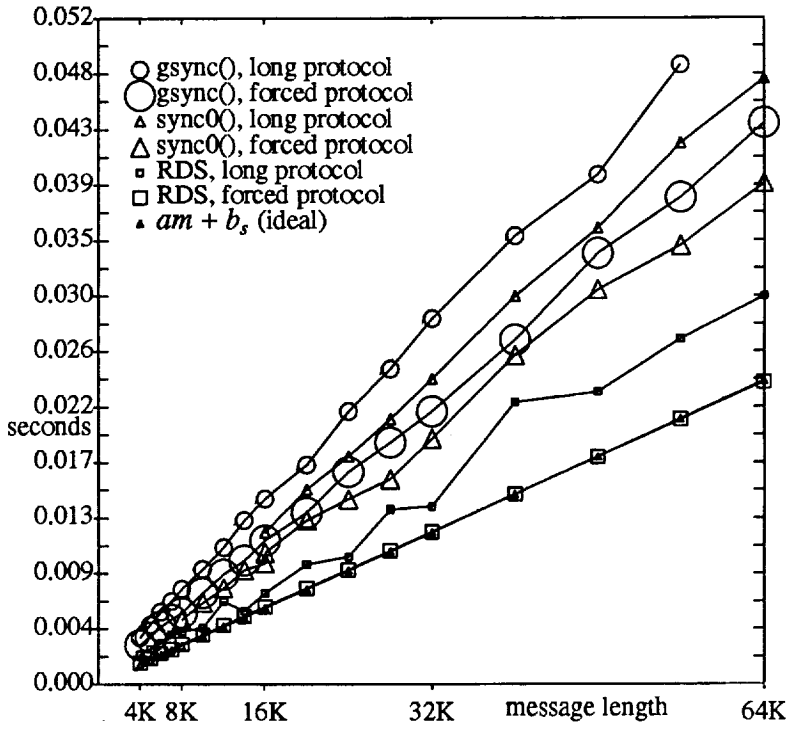

Fig. 4-2: Cost of shifts in a 32-node ring on the iPSC/860.

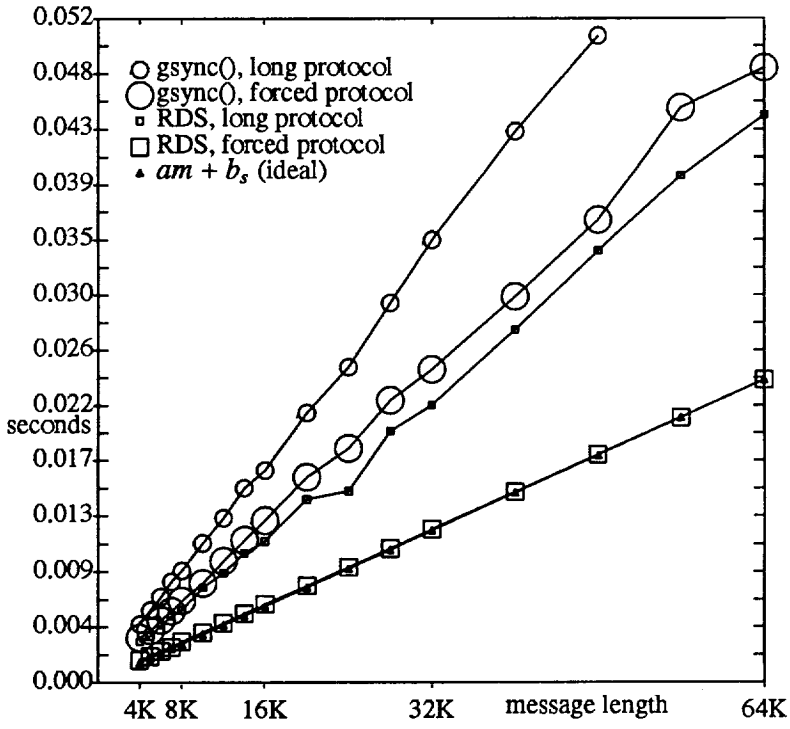

Fig. 4-3: Cost of shifts in a 128-node ring on the iPSC/860.

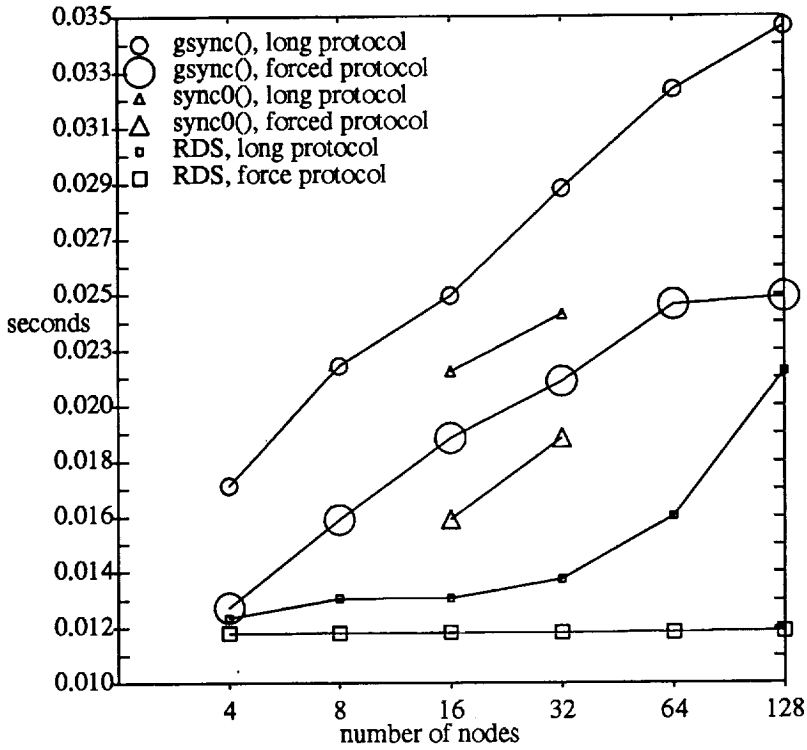

Fig. 4-4: Average time of shifting 32K byte messages in a ring.

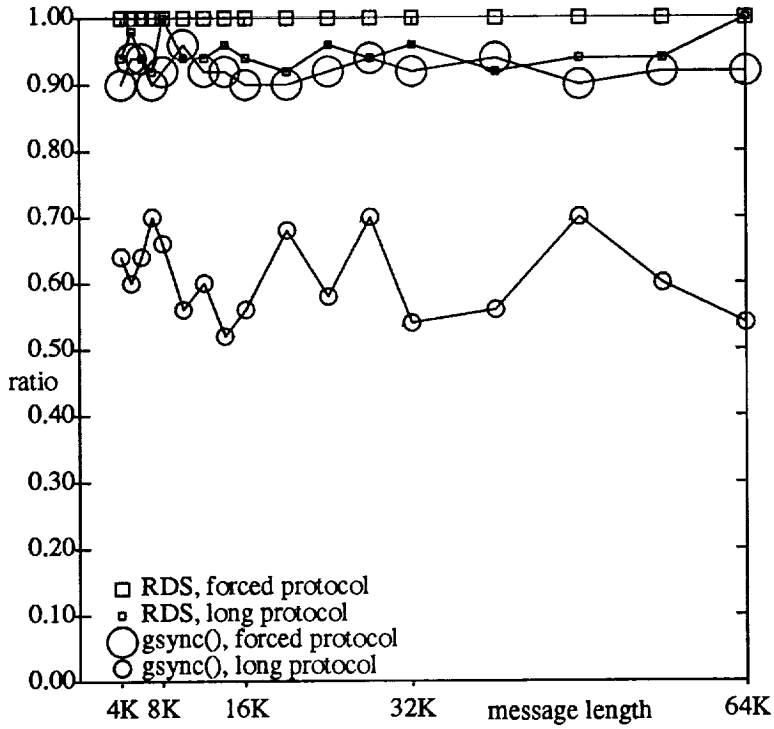

Fig. 4-5: Successful concurrent shifts for 50 trials
in a 4-node ring on the iPSC/860.

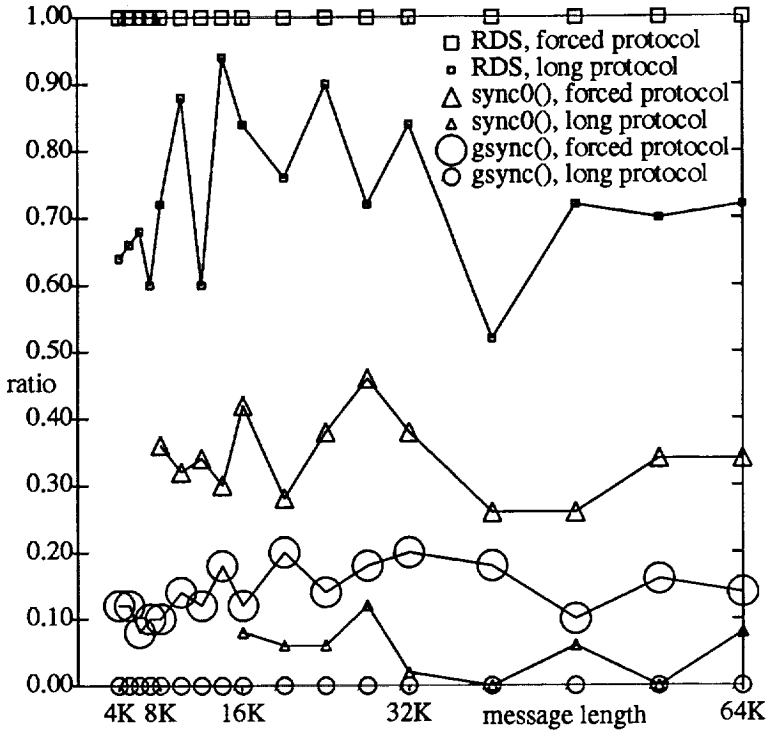

Fig. 4-6: Successful concurrent shifts for 50 trials in a 32-node ring on the iPSC/860.

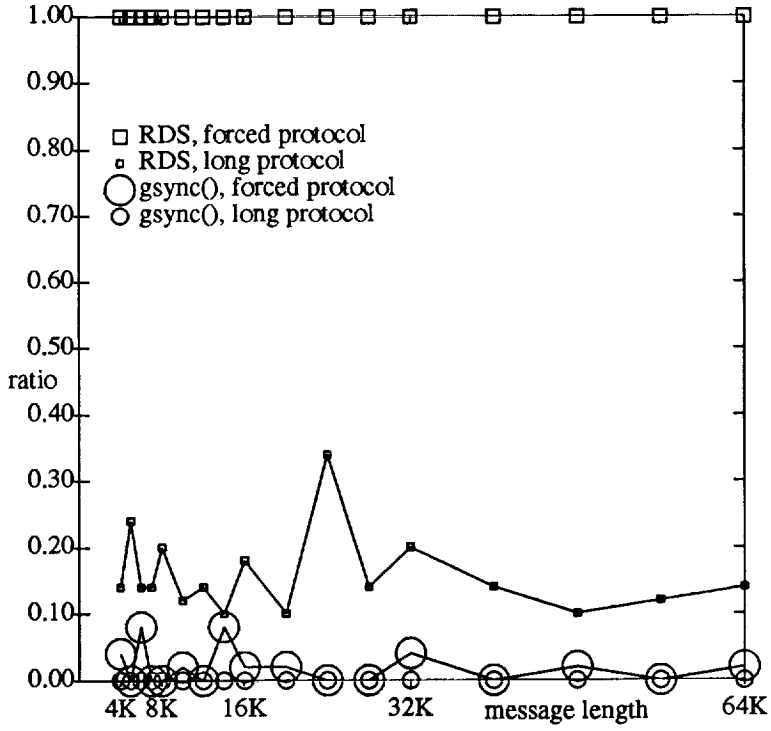

Fig. 4-7: Successful concurrent shifts for 50 trials in a 128-node ring on the iPSC/860.