NASA Contractor Report 187498
ICASE Report No. 91-4

ICASE

COMPLETE EXCHANGE ON THE iPSC-860

Shahid H. Bokhari

Contract No. NAS1-18605
January 1991

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665-5225

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Complete Exchange on the iPSC-860*

Shahid H. Bokhari

Department of Electrical Engineering

University of Engineering & Technology, Lahore, Pakistan

and

ICASE, NASA Langley Research Center

Hampton, Virginia

Abstract

The implementation of complete exchange on the circuit switched Intel iPSC-860 hypercube is described. This pattern, also known as all-to-all personalized communication, is the densest requirement that can be imposed on a network. On the iPSC-860, care needs to be taken to avoid edge contention, which can have a disastrous impact on communication time. There are basically two classes of algorithms that achieve contention-free complete exchange. The first contains the classical standard exchange algorithm that is generally useful for small message sizes. The second includes a number of optimal or near-optimal algorithms that are best for large messages.

Measurements of communication overhead on the iPSC-860 are given and a notation for analyzing communication link usage is developed. It is shown that for the two classes of algorithms, there is substantial variation in performance with synchronization technique and choice of message protocol. Timings of six implementations are given; each of these is useful over a particular range of message size and cube dimension.

Since the complete exchange is a superset of all communication patterns, these timings represent upper bounds on the time required by an arbitrary communication requirement. These results indicate that the programmer needs to evaluate several possibilities before finalizing an implementation—a careful choice can lead to very significant savings in time.

*Research supported by the National Aeronautics and Space Administration under NASA contract NAS1-18605 while the author was in residence at the Institute for Computer Applications in Science & Engineering, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665-5225.

1 Introduction

The complete exchange (all-to-all personalized) communication pattern arises in many important parallel processing applications. Some examples are matrix transpose, matrix-vector multiply, certain implementations of the 2-dimensional FFT and distributed table lookup. The complete exchange is equivalent to the complete directed graph and is, as such, the densest communication requirement that can be imposed on a network.

We describe the implementation of this pattern on the circuit switched Intel iPSC-860 hypercube. There are two basic classes of algorithms for complete exchange on this machine. The first class contains the well known standard exchange store-and-forward algorithm that uses, for a d-dimensional machine, d messages of size 2^{d-1} blocks each and is useful when the blocks are small. The second class is made up of a number of optimal or near-optimal circuit-switched algorithms that use 2^d or $2^d - 1$ messages of size 1 block each and are always better than standard exchange for large enough block size. Within each class there is substantial variation in performance with synchronization technique and choice of message protocol. Consequently there are no less than six implementations, each of which is useful for some range of message size and hypercube dimension.

In Section 2 of this paper we describe the interconnection network, routing strategy and communication performance of the Intel iPSC-860. We show the impact of message protocol and distance on communication time and also show how edge contention can be disastrous. We discuss the complete exchange pattern in Section 3 and introduce a tabular notation for its communication link requirements. This is helpful in presenting the various algorithms that follow in Section 4.

Details of our implementations are given in Section 5. In Section 6 we present measured timings of the six implementations. We conclude with Section 7, which contains a discussion of our results and speculations on how our observations on complete exchange apply to arbitrary communication patterns.

2 The Intel iPSC-860 hypercube

The interconnection network of a 32 node hypercube is shown in Figure 1. The labeled vertices hanging from each vertex of the network represent processors of the hypercube. Two processors in the network are connected if and only if the binary representations of their labels differ in exactly one bit. An important feature of interprocessor communications in the Intel hypercube is circuit switching. When two nodes wish to communicate, a dedicated path is set up between them. Messages then flow through this path without involving intervening processors. The path between source and destination is determined by the 'e-cube' routing algorithm: starting with the right hand side of the binary label of the source processor, we move to the processor whose label most closely matches the label of the destination processor.

Since the routing algorithm is fixed, we can encounter *edge* and *node* contention. Edge contention is the sharing of an edge (i.e. a communication link) by two or more paths. Similarly, node contention is the sharing of a node.

Figure 1 *illustrates* paths from 0 to 31 (solid), 2 to 23 (dashed) and 14 to 11 (dotted). The *lengths* of these paths (the *distance* between source and destination) are 5, 3 and 2 respectively. The paths $0 \rightarrow 31$ and $2 \rightarrow 23$ share the edge 3–7, while the paths $0 \rightarrow 31$ and $14 \rightarrow 11$ share node 15.

2.1 Measurements of Communication Overhead

We now provide measurements of the communication overhead on the iPSC-860. Earlier but more extensive measurements are given in [1]. Detailed analyses of some aspects of the communication system on the Intel iPSC-2* and on the iPSC-860 appear in [6, 7, 8].

2.2 Impact of path length

There are two message types (selectable by the programmer) on the iPSC-860[3]. A message of the FORCED type is discarded upon arrival if no receive

*The iPSC-2 is an earlier hypercube that uses less powerful 80386 processors but has interconnection hardware similar to the iPSC-860.

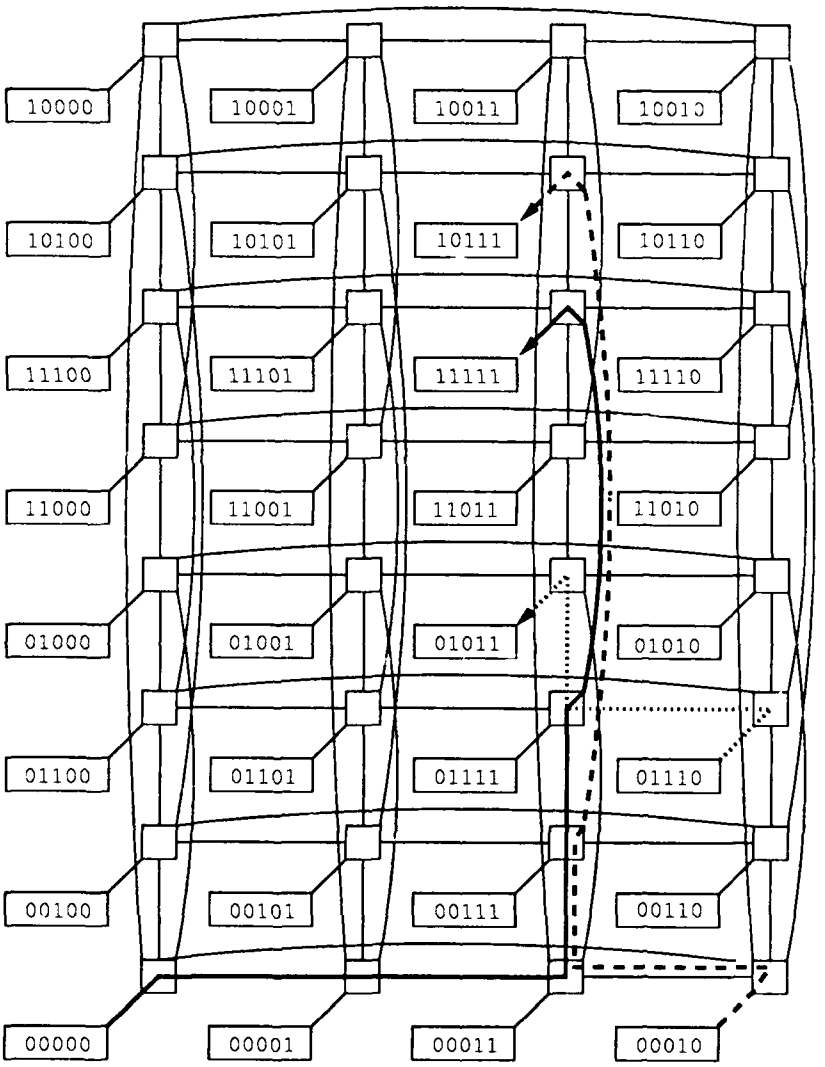


Figure 1: Interconnection network of a 32 node hypercube.

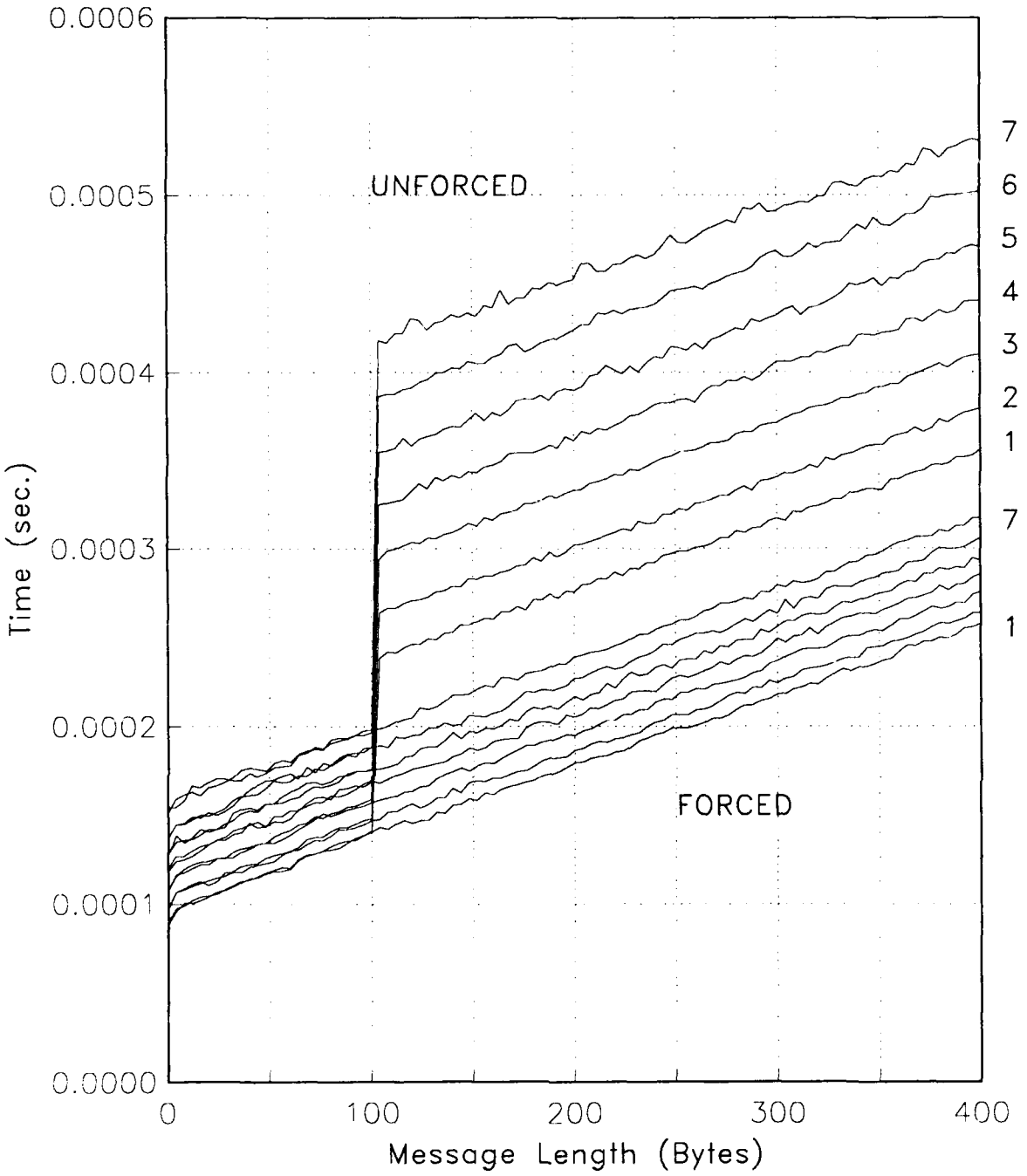


Figure 2: Impact of path length: 0–400 bytes.

has been posted for it. A message of the UNFORCED type is stored in a system buffer if it arrives and no receive has been posted for it. The performance of both types is similar for messages of size 0–100 bytes. Beyond 100 bytes, an UNFORCED message is preceded by the exchange of 'reserve-acknowledge' messages that cause space to be reserved in the destination. This process causes significant overhead as we shall see in what follows.

Figure 2 shows the time required to communicate messages of length 0–400 bytes between processors that are $1, 2, \cdots, 7$ communication links apart. The time required to send a 0 byte message to a neighboring node (i.e. distance 1 away) is about 95 μsec. (this represents the absolute minimum for any communication operation on this machine). The time to communicate a 0 byte message over the maximum distance of 7 is 155 μsec. Inspection of these plots reveals that they are linear, parallel and evenly distributed from 0 to 100 bytes. The communication time increases at about 10 μsec. per communication link. This is a far from negligible variation: the time required to send a 4-byte floating point number distance 7 away is nearly double the time to send it to a neighboring node.[†]

At message length 101 bytes our curves bifurcate into two families. The lower family represents FORCED messages and the upper family UNFORCED messages. The separation is due to the overhead of the reservation messages described above.

The time (in μsec.) to communicate a message of length m bytes over distance d is $t = 95 + 0.394m + 10.3d$ for FORCED messages. The times for UNFORCED messages is identical for $0 < m \leq 100$ and is $t = 164 + 0.398m + 29.9d$ for $m > 100$. The time for zero byte messages on the plots is slightly below what would be predicted by these expressions.

2.3 Impact of edge contention

The contention experiment uses 8 source-destination pairs that are depicted in Figure 3. The e-cube algorithm generates the following paths.

[†]The strategy for communication used here is as follows. A receive is posted on the destination processor, followed by a global synchronization. The source processor then sends a message and waits for the destination to signal receipt via another global synchronization. The synchronization times are not included in the plotted data. This strategy leads to timings that are different from those reported in [1] and elsewhere, but accurately represent the state of affairs in our implementations of complete exchange.

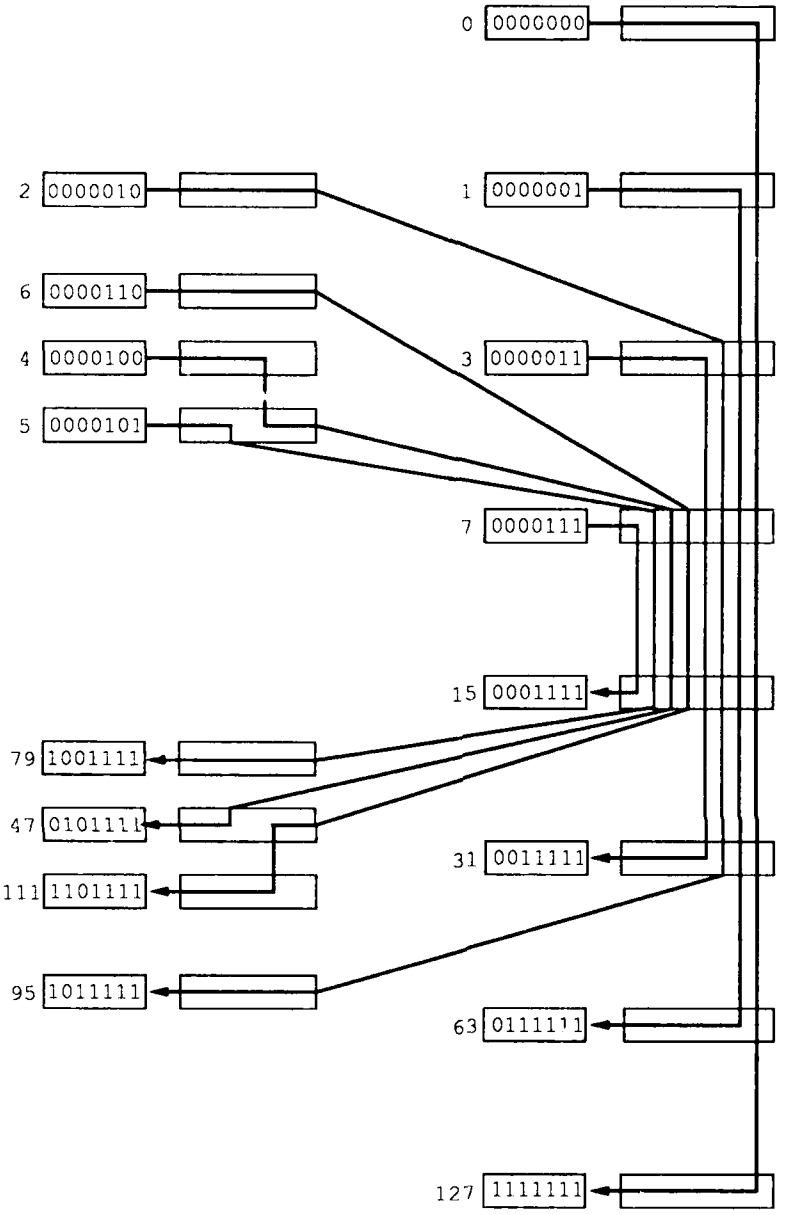


Figure 3: Communication pattern for edge contention experiment.

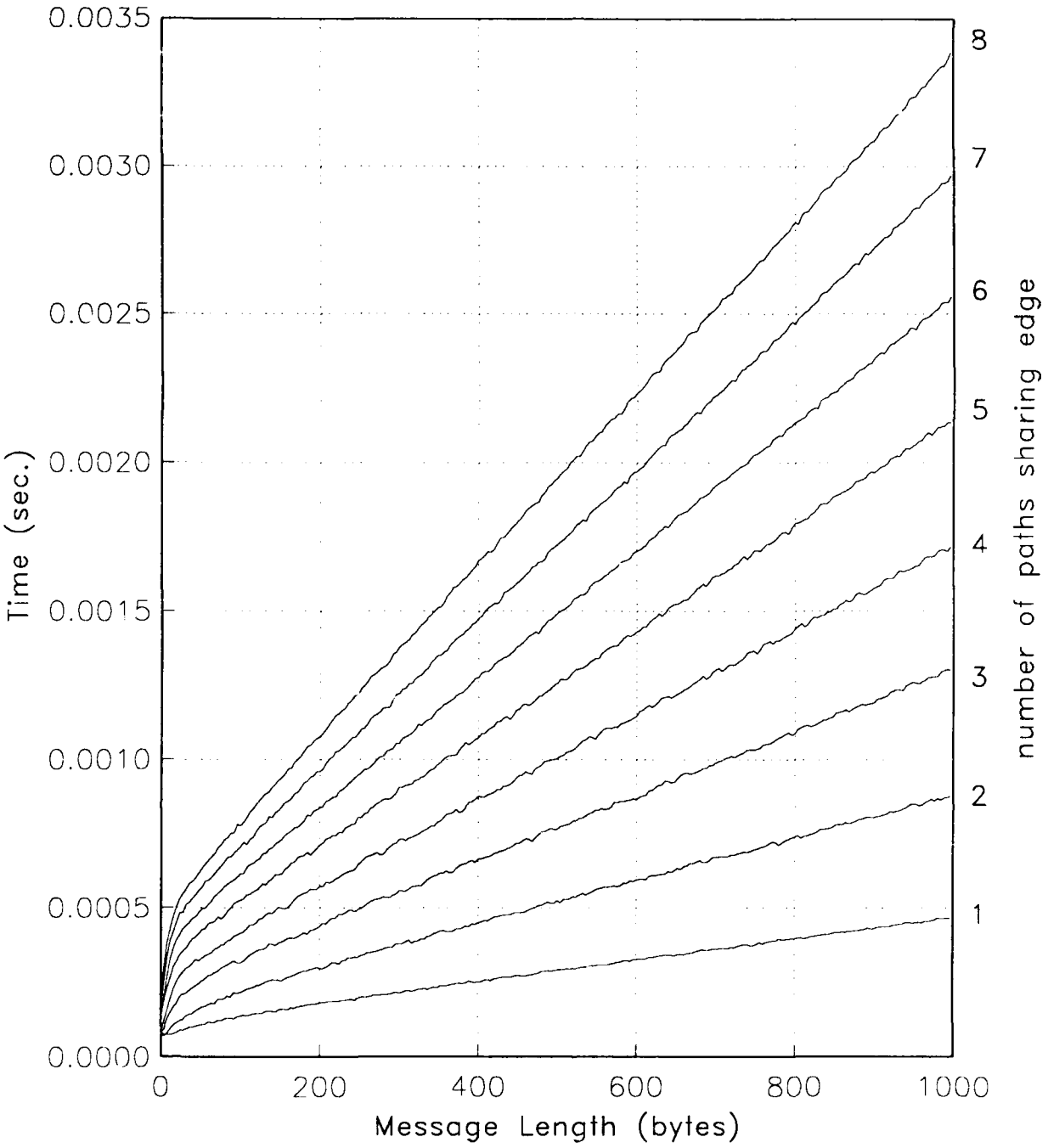


Figure 4: Observations from contention experiment.

$$0 \rightarrow 1 \rightarrow 3 \rightarrow 7 \rightarrow 15 \rightarrow 31 \rightarrow 63 \rightarrow 127 \quad (1)$$
$$1 \rightarrow 3 \rightarrow 7 \rightarrow 15 \rightarrow 31 \rightarrow 63 \quad (2)$$
$$3 \rightarrow 7 \rightarrow 15 \rightarrow 31 \quad (3)$$
$$7 \rightarrow 15 \quad (4)$$
$$5 \rightarrow 7 \rightarrow 15 \rightarrow 79 \quad (5)$$
$$6 \rightarrow 7 \rightarrow 15 \rightarrow 47 \quad (6)$$
$$2 \rightarrow 3 \rightarrow 7 \rightarrow 15 \rightarrow 31 \rightarrow 95 \quad (7)$$
$$4 \rightarrow 5 \rightarrow 7 \rightarrow 15 \rightarrow 47 \rightarrow 111 \quad (8)$$

This experiment has been designed to impose the maximum possible amount of contention on one edge. Thus we have 8 paths sharing edge $7 \rightarrow 15$. This experiment was run for message lengths of 0 to 1000 bytes using FORCED types.

Figure 4 shows a set of line plots from the contention experiment. The plot labeled 1 shows the time for path (1) ($0 \rightarrow 127$) alone, plot 2 shows the time for paths (1) and (2) simultaneously and so on. The time required by eight contending messages of 1000 bytes is more than seven times the time required for one message.

In contrast with *edge* contention, which has disastrous impact on communication time, *node* contention has no measurable impact on the iPSC-860.

The Intel iPSC-2 and IPSC-860 are among the first commercial examples of circuit-switched machines. Since circuit switching provides very fast communications, it is generally felt that it eliminates most of the inefficiencies caused by communication overhead. In particular, it is a common belief that programmers can ignore the details of the interconnection network. This is a mistaken belief since, as we shall see later in this paper, very careful consideration of the interconnection is necessary if the full power of the machine is to be utilized.

3 The Complete Exchange Pattern

When executing the complete exchange on a distributed memory parallel machine, each of n processors must send a different block to each of the

remaining $n-1$ processors. The complete exchange is required by many important algorithms. These include the Alternating Directions Implicit (ADI) method, which makes heavy use of the matrix transpose which in turn is essentially equivalent to the complete exchange. Other important examples include matrix-matrix and matrix-vector multiply, certain implementations of the 2-*d* FFT and distributed table lookup.

The complete exchange is equivalent to the complete directed graph and, as such, is the densest communication requirement that can be imposed on a network. Any arbitrary communication pattern must necessarily be a subset of the complete exchange. The time for the complete exchange on a given machine is an upper bound on the time for an arbitrary communication requirement.

Because of its generality and widespread applications, it is worthwhile to investigate the time required to execute this pattern and to develop fast procedures for it, as we proceed to do in the following Sections. At this point we present a chart (Figure 5) that depicts the communications requirements of the transpose algorithm under the 'e-cube' routing algorithm on a hypercube.

The row labels in Figure 5 represent source-destination ordered pairs. Thus 010→110 represents the transmission of a message from processor 2 to processor 6. The column labels represent communication links or edges. These are in groups of 3 to indicate sets of links emanating from a single node. Thus the group $\underline{\underline{\underline{010}}}$ indicates the 3 links emanating from node 2. The *rightmost* of these is the link that connects 010 to 011, the middle link is the one connecting 010 to 000 and the leftmost is the link connecting 010 to 110.

An x in this chart at position $< row, column >$ indicates that the edge corresponding to *column* is used by the e-cube routing algorithm when transmitting from the source to the destination that specifies the *row*. A dot indicates that the corresponding edge is not used. Thus the row labeled 010→101 has x's in the columns $\underline{\underline{010}}$, $\underline{\underline{011}}$ and $\underline{\underline{001}}$, corresponding to the e-cube route 010→011→001→101.

It is an easily verified property of the e-cube routing algorithm that no message originating in a node whose label's leftmost binary bit is a 0 (1) can use an edge that lies between two nodes whose labels' leftmost bits are both 1 (0). This permits us to omit half the edges from each row in Figure 5 and make the chart compact. The two sets of rows in Figure 5 represent two sets of edge disjoint paths.

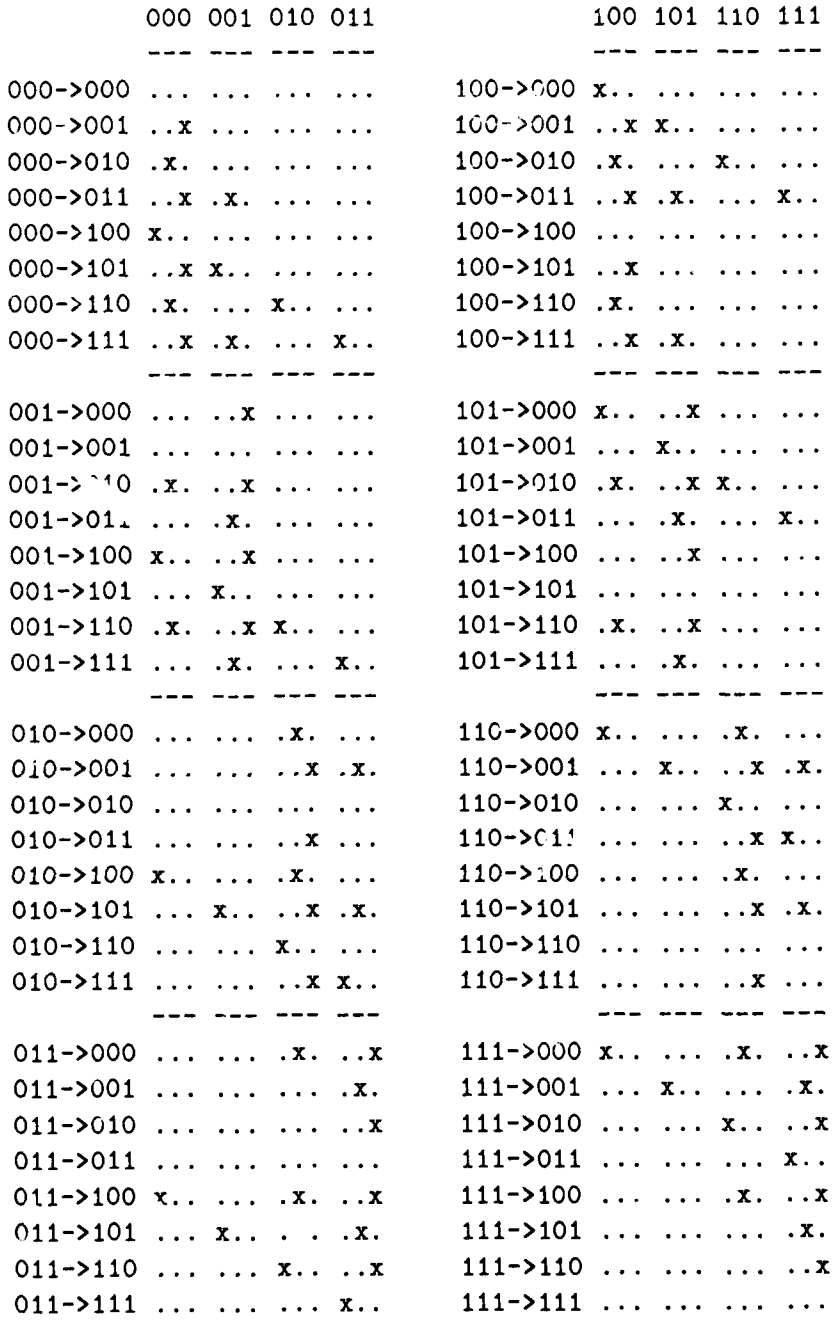


Figure 5: Link usage of the complete exchange communication pattern on an 8-node hypecube

Every algorithm for complete exchange that transmits one block at a time generates a schedule for the transmissions shown in the chart of Figure 5. This schedule is simply a numbering of the x's in the chart under the following constraints.

Constraint	Significance
Two x's in the same column cannot have the same number.	A link cannot be used for the transmission of two messages simultaneously.
All x's in the same row must have the same number.	All links in an e-cube routed chain are in use simultaneously because of circuit switching.
Two rows in a group (e.g the group 000->... in Figure 5) cannot have the same number.	A processor can only transmit on one link at a time.

The largest number in any schedule determines the time required (in block transmissions) to execute the complete exchange. This cannot be less than $n-1$ because each processor must send out $n-1$ blocks serially.

4 Algorithms for Complete Exchange

A naive algorithm for complete exchange can be described as follows: at time step i, each processor sends out the block destined for processor i. The program that executes in each processor is as follows.

```
procedure naive;
begin
    for destination = 0 to n − 1 do
        if (destination ≠ mynumber) {no need to send to myself}
            send_block_to_processor(destination);
end
```

```
          000 001 010 011             100 101 110 111
          --- --- --- ---             --- --- --- ---
000->000  ... ... ... ...   100->000 1.. ... ... ...
000->001  ..1 ... ... ...   100->001 ..2 2.. ... ...
000->010  .2. ... ... ...   100->010 .3. ... 3.. ...
000->011  ..3 .3. ... ...   100->011 ..4 .4. ... 4..
000->100  4.. ... ... ...   100->100 ... ... ... ...
000->101  ..5 5.. ... ...   100->101 ..5 ... ... ...
000->110  .6. ... 6.. ...   100->110 .6. ... ... ...
000->111  ..7 .7. ... 7..   100->111 ..7 .7. ... ...
          --- --- --- ---             --- --- --- ---
001->000  ... ..1 ... ...   101->000 2.. ..2 ... ...
001->001  ... ... ... ...   101->001 ... 3.. ... ...
001->010  .3. ..3 ... ...   101->010 .4. ..4 4.. ...
001->011  ... .4. ... ...   101->011 ... .5. ... 5..
001->100  5.. ..5 ... ...   101->100 ... ..6 ... ...
001->101  ... 6.. ... ...   101->101 ... ... ... ...
001->110  .7. ..7 7.. ...   101->110 .7. ..7 ... ...
001->111  ... .8. ... 8..   101->111 ... .8. ... ...
          --- --- --- ---             --- --- --- ---
010->000  ... ... .1. ...   110->000 3.. ... .3. ...
010->001  ... ... ..2 .2.   110->001 ... 4.. ..4 .4.
010->010  ... ... ... ...   110->010 ... ... 5.. ...
010->011  ... ... ..3 ...   110->011 ... ... ..6 6..
010->100  6.. ... .6. ...   110->100 ... ... .7. ...
010->101  ... 7.. ..7 .7.   110->101 ... ... ..8 .8.
010->110  ... ... 8.. ...   110->110 ... ... ... ...
010->111  ... ... ..9 9..   110->111 ... ... ..9 ...
          --- --- --- ---             --- --- --- ---
011->000  ... ... .2. ..2   111->000 4.. ... .4. ..4
011->001  ... ... ... .3.   111->001 ... 5.. ... .5.
011->010  ... ... ... ..4   111->010 ... ... 6.. ..6
011->011  ... ... ... ...   111->011 ... ... ... 7..
011->100  7.. ... .7. ..7   111->100 ... ... .8. ..8
011->101  ... 8.. ... .8.   111->101 ... ... ... .9.
011->110  ... ... 9.. ..9   111->110 ... ... ... ..a
011->111  ... ... ... a..   111->111 ... ... ... ...
```

Figure 6: Schedule generated by the naive algorithm

A little reflection reveals that this algorithm concentrates traffic on the links entering node i during time step i. As a result we expect to see contention for links and hence poor performance. Figure 6 shows what can happen. This Figure is the chart of Figure 5 with the x's replaced by numbers to indicate the time step during which a link is utilized, as described in the previous Section.

In Figure 6 we have assumed that link contention is resolved by granting a path to the lowest numbered processor. It is possible to demonstrate that the time required under this assumption (i.e. the highest number in the schedule, 10 in Figure 6) is $\frac{3}{2}n - 2$. Since optimal ($n - 1$ step) algorithms are known, the naive algorithm only serves to show how poor a bad approach can be.[‡] Careless programmers have, nevertheless, been known to use this algorithm in practice.

4.1 Two $n - 1$ step Optimal Algorithms

The contention for links that disrupts transmissions can be eliminated by careful scheduling. A simple algorithm that achieves this is `linear`.

```
procedure linear;
begin
    for i = 1 to n − 1 do
        send_block_to_processor((mynumber + i)mod(n));
end
```

The schedule generated by this algorithm is shown in Figure 7. It is easy to verify that this schedule takes exactly $n - 1$ steps with no contention.

Seidel et al. have studied the iPSC-2 and iPSC-860's communication system in great detail. They have shown [6, 7, 8] that, under certain circumstances, it is preferable to decompose a communication requirement into pairwise exchanges. Their research shows that this can lead to great savings in communication time. A schedule for the complete exchange that is composed of only pairwise exchanges and takes exactly $n - 1$ steps is given in [7]. This is described as follows.

[‡]However $\frac{3}{2}n - 2$ is much better than the $O(n \log n)$ standard exchange algorithm for large n, as can be verified by measurements.

```
          000 001 010 011              100 101 110 111
          --- --- --- ---              --- --- --- ---
000->000  ... ... ... ...    100->000  4.. ... ... ...
000->001  ..1 ... ... ...    100->001  ..5 5.. ... ...
000->010  .2. ... ... ...    100->010  .6. ... 6.. ...
000->011  ..3 .3. ... ...    100->011  ..7 .7. ... 7..
000->100  4.. ... ... ...    100->100  ... ... ... ...
000->101  ..5 5.. ... ...    100->101  ..1 ... ... ...
000->110  .6. ... 6.. ...    100->110  .2. ... ... ...
000->111  ..7 .7. ... 7..    100->111  ..3 .3. ... ...
          --- --- --- ---              --- --- --- ---
001->000  ... ..7 ... ...    101->000  3.. ..3 ... ...
001->001  ... ... ... ...    101->001  ... 4.. ... ...
001->010  .1. ..1 ... ...    101->010  .5. ..5 5.. ...
001->011  ... .2. ... ...    101->011  ... .6. ... 6..
001->100  3.. ..3 ... ...    101->100  ... ..7 ... ...
001->101  ... 4.. ... ...    101->101  ... ... ... ...
001->110  .5. ..5 5.. ...    101->110  .1. ..1 ... ...
001->111  ... .6. ... 6..    101->111  ... .2. ... ...
          --- --- --- ---              --- --- --- ---
010->000  ... ... .6. ...    110->000  2.. ... .2. ...
010->001  ... ... ..7 .7.    110->001  ... 3.. ..3 .3.
010->010  ... ... ... ...    110->010  ... ... 4.. ...
010->011  ... ... ..1 ...    110->011  ... ... ..5 5..
010->100  2.. ... .2. ...    110->100  ... ... .6. ...
010->101  ... 3.. ..3 .3.    110->101  ... ... ..7 .7.
010->110  ... ... 4.. ...    110->110  ... ... ... ...
010->111  ... ... ..5 5..    110->111  ... ... ..1 ...
          --- --- --- ---              --- --- --- ---
011->000  ... ... .5. ..5    111->000  1.. ... .1. ..1
011->001  ... ... ... .6.    111->001  ... 2.. ... .2.
011->010  ... ... ... ..7    111->010  ... ... 3.. ..3
011->011  ... ... ... ...    111->011  ... ... ... 4..
011->100  1.. ... .1. ..1    111->100  ... ... .5. ..5
011->101  ... 2.. ... .2.    111->101  ... ... ... .6.
011->110  ... ... 3.. ..3    111->110  ... ... ... ..7
011->111  ... ... ... 4..    111->111  ... ... ... ...
```

Figure 7: Schedule generated by the linear optimal algorithm

```
procedure pairwise;
begin
    for i = 1 to n − 1 do
        send_block_to_processor(mynumber ⊕ i);
end
```

Figure 8 gives the schedule generated by this algorithm for an 8 node hypercube. The format of this chart is different from the preceding charts. This is because under pairwise decomposition, our problem becomes one of scheduling an undirected graph. Each edge of this graph represents a pairwise exchange between the two nodes at its endpoints. The exchange between 000 and 101 means that the edges 000->001,001->101 are occupied at the same time that the edges 101->100,100->000 are occupied. As is usual with undirected graphs, only $n(n-1)/2$ rows need to be specified.

Figure 8 illustrates that (1) procedure `pairwise` decomposes the complete exchange into pairwise exchanges, (2) no two exchanges use the same link during the same time step, and (3) the total number of steps required is 7 (in general it is $n-1$).

4.2 An n step Stable Algorithm

For the linear and pairwise algorithms to function correctly, all communication steps must start in synchrony. The iPSC-860 is not an SIMD machine and does not have a master clock or a central instruction issue unit. As such, we can expect some slight drift in the absolute times at which transmissions are initiated. This is caused by drifts in the individual clocks of the the processors as well as by unpredictable operating system overhead. This drift can be eliminated by using explicit synchronization before each transmission, a solution that incurs substantial overhead.

The stable algorithm has been designed to tolerate considerable drift in the timings of the transmissions and does not need synchronization before each transmission.

```
          000 001 010 011 100 101 110 111
          --- --- --- --- --- --- --- ---
000--001  ..1 ..1 ... ... ... ... ... ...
000--010  .2. ... .2. ... ... ... ... ...
000--011  ..3 .3. .3. ..3 ... ... ... ...
000--100  4.. ... ... ... 4.. ... ... ...
000--101  ..5 5.. ... ... 5.. ..5 ... ...
000--110  .6. ... 6.. ... 6.. ... .6. ...
000--111  ..7 .7. ... 7.. 7.. ... .7. ..7
          --- --- --- --- --- --- --- ---
001--010  .5. ..5 ..5 .5. ... ... ... ...
001--011  ... .2. ... .2. ... ... ... ...
001--100  6.. ..6 ... ... ..6 6.. ... ...
001--101  ... 3.. ... ... ... 3.. ... ...
001--110  .7. ..7 7.. ... ... 7.. ..7 .7.
001--111  ... .4. ... 4.. ... 4.. ... .4.
          --- --- --- --- --- --- --- ---
010--011  ... ... ..6 ..6 ... ... ... ...
010--100  3.. ... .3. ... .3. ... 3.. ...
010--101  ... 7.. ..7 .7. .7. ..7 7.. ...
010--110  ... ... 4.. ... ... ... 4.. ...
010--111  ... ... ..1 1.. ... ... 1.. ..1
          --- --- --- --- --- --- --- ---
011--100  7.. ... .7. ..7 ..7 .7. ... 7..
011--101  ... 4.. ... .4. ... .4. ... 4..
011--110  ... ... 1.. ..1 ... ... ..1 1..
011--111  ... ... ... 5.. ... ... ... 5..
          --- --- --- --- --- --- --- ---
100--101  ... ... ... ... ..1 ..1 ... ...
100--110  ... ... ... ... .5. ... .5. ...
100--111  ... ... ... ... ..2 .2. .2. ..2
          --- --- --- --- --- --- --- ---
101--110  ... ... ... ... .2. ..2 ..2 .2.
101--111  ... ... ... ... ... .6. ... .6.
          --- --- --- --- --- --- --- ---
110--111  ... ... ... ... ... ... ..3 ..3
```

Figure 8: Schedule generated by the pairwise algorithm.

```
procedure stable;
begin
    for i = 0 to n − 1 do
        begin
            if(mynumber < n/2)
                destination = (mynumber × 2 + 1 + i)mod(n)
            else
                destination = (mynumber × 2 − n + i)mod(n);
            if (destination = mynumber)
                idle
            else
                send_block_to_processor(destination);
        end;
end
```

It can be seen in Figure 9 that no column has two consecutive integers in it. As an example, consider the transmission 010->111, which uses the edges 010->011 and 011->111 in time step 3. These links are not used again by another processor until step 5 (when the transmission 001->111 uses 011->111). Thus the drift of a full transmission period can be tolerated by this schedule. The price of this stability is an increase in the total time from $n-1$ to n. It is impossible to obtain a stable schedule of length $n-1$ since an odd number of time periods cannot have the stability property (n is an even number, since we are dealing with hypercubes).

4.3 The Standard Exchange Algorithm

The standard exchange procedure [4] uses $\log n$ transmissions of size $n/2$ blocks each. All transmissions are along paths of length 1, thus there is no possibility of contention. This algorithm incurs massive overhead (1) because of the perfect shuffling of blocks and (2) because each processor transmits a total of $\frac{n}{2}\log n$ blocks, rather than n or $n-1$ blocks for the the algorithms discussed above. It is, nevertheless, competitive for small block sizes since there are only $\log n$ transmissions (as opposed to n or $n-1$ for the abovementioned algorithms) and thus the overhead of starting up a message

```
          000 001 010 011              100 101 110 111
          --- --- --- ---              --- --- --- ---
000->000  ... ... ... ...    100->000  1.. ... ... ...
000->001  ..1 ... ... ...    100->001  ..2 2.. ... ...
000->010  .2. ... ... ...    100->010  .3. ... 3.. ...
000->011  ..3 .3. ... ...    100->011  ..4 .4. ... 4..
000->100  4.. ... ... ...    100->100  ... ... ... ...
000->101  ..5 5.. ... ...    100->101  ..6 ... ... ...
000->110  .6. ... 6.. ...    100->110  .7. ... ... ...
000->111  ..7 .7. ... 7..    100->111  ..8 .8. ... ...
          --- --- --- ---              --- --- --- ---
001->000  ... ..6 ... ...    101->000  7.. ..7 ... ...
001->001  ... ... ... ...    101->001  ... 8.. ... ...
001->010  .8. ..8 ... ...    101->010  .1. ..1 1.. ...
001->011  ... .1. ... ...    101->011  ... .2. ... 2..
001->100  2.. ..2 ... ...    101->100  ... ..3 ... ...
001->101  ... 3.. ... ...    101->101  ... ... ... ...
001->110  .4. ..4 4.. ...    101->110  .5. ..5 ... ...
001->111  ... .5. ... 5..    101->111  ... .6. ... ...
          --- --- --- ---              --- --- --- ---
010->000  ... ... .4. ...    110->000  5.. ... .5. ...
010->001  ... ... ..5 .5.    110->001  ... 6.. ..6 .6.
010->010  ... ... ... ...    110->010  ... ... 7.. ...
010->011  ... ... ..7 ...    110->011  ... ... ..8 8..
010->100  8.. ... .8. ...    110->100  ... ... .1. ...
010->101  ... 1.. ..1 .1.    110->101  ... ... ..2 .2.
010->110  ... ... 2.. ...    110->110  ... ... ... ...
010->111  ... ... ..3 3..    110->111  ... ... ..4 ...
          --- --- --- ---              --- --- --- ---
011->000  ... ... .2. ..2    111->000  3.. ... .3. ..3
011->001  ... ... ... .3.    111->001  ... 4.. ... .4.
011->010  ... ... ... ..4    111->010  ... ... 5.. ..5
011->011  ... ... ... ...    111->011  ... ... ... 6..
011->100  6.. ... .6. ..6    111->100  ... ... .7. ..7
011->101  ... 7.. ... .7.    111->101  ... ... ... .8.
011->110  ... ... 8.. ..8    111->110  ... ... ... ..1
011->111  ... ... ... 1..    111->111  ... ... ... ...
```

Figure 9: Schedule generated by the stable algorithm

(95 μsec per transmission, see Section 2.1) is not incurred as frequently.

```
procedure standard;
begin
   for j = d − 1 downto 0 do
      begin
         if (bit j of mynumber = 0) then
            message= blocks (n/2) to n − 1;
         else
            message= blocks 0 to (n/2) − 1;
         destination = mynumber ⊕ 2^j;
         send_message_to_processor(destination);
         shuffle blocks;
      end;
end
```

5 Implementation Details

We now briefly discuss the relevant details of our implementations of the algorithms discussed in the preceding Section. After considerable experimentation we have identified six implementations that are useful in the sense that each one of them outperforms all others for some values of hypercube dimension and block size.

The **message type** used is one factor to be considered when implementing an an algorithm on the iPSC-860. The distinction between FORCED and UNFORCED types has already been discussed in Section 2.1. UNFORCED types are not competitive beyond 100 bytes because of the overhead of the "reserve-acknowledge" cycle. For messages up to 100 bytes in size they can sometimes lead to better performance.

The **synchronization technique** is another important factor. When using FORCED message types it is essential for each processor to post receives for all expected messages in the procedure at the very beginning, and to carry out a global synchronization after this. Omission of the (expensive) global synchronization step is fatal as it leads to messages arriving before their corresponding receives have been posted and thus being discarded by the operating system. When using UNFORCED messages, it is possible to omit this

global synchronization step since these messages are stored by the operating system until the required receive has been posted. The programmer must, however, be careful to ensure that there is enough free memory available to the operating system so that buffers can be allocated for all messages that may arrive without posted receives.

Finally, the issue of **pairwise exchanges** arises because of an idiosyncrasy of the iPSC's communication hardware. A receive and a transmit occurring nearly simultaneously at a processor can proceed concurrently, while a short delay causes them to be carried out serially. This issue has been researched in detail by Seidel et al. [6, 7, 8]. It has been shown that a pairwise exchange is guaranteed to proceed concurrently if the two processors involved first exchange a pair of zero byte "pairwise synchronization" messages. The time for this pairwise synchronization is far less than the time for global synchronization and is negligible for moderate to large messages.

For the `linear` and `stable` algorithms we use `FORCED` message types and post all receives before a global synchronization step. A complicating factor in the `stable` algorithm (Section 4.2) is the need for an idle period. To ensure the correct operation of the stable scheme, this period must be equal to the time taken for transmission by non-idling processors. This is achieved by busy waiting for a period given by the expression for transmission time of Section 2.2.

There are two implementations each for `pairwise` and `standard`. We use `FORCED` types in `pairwise_F` and `standard_F` and post all receives before a global synchronization. In addition, we use the pairwise synchronization technique of [6, 7][§]. The second pair of implementations `pairwise_U` and `exchange_U` uses `UNFORCED` types with no synchronization. The following table summarizes the details of our implementations.

[§] We post all receives for the pairwise synchronization messages before the global synchronization. This results in better performance than the method proposed in [7] which does not use global synchronization.

IMPLEMENTATION	SYNCHRONIZATION	MESSAGE TYPE
`linear`	Global	FORCED
`stable`		
`pairwise_F`	Global and pairwise	
`standard_F`		
`pairwise_U`	None	UNFORCED
`standard_U`		

6 Experimental Observations

Figures 10 & 11 show the times for all six implementations against block sizes for Intel iPSC-860 hypercubes of dimension 1-7. For $d = 6$ & 7, each point in the plots is the *maximum* of 100 observations; for $d = 1 \cdots 5$, each point is the maximum of 1000 observations. It is important to evaluate these implementations with respect to their maximum run times, rather than average or minimum. This is because there is enormous variability in the run times of implementations that do not use pairwise synchronization (see Section 6.4, below) and a comparison based on average or minimum run times would be misleading.

`linear`, `stable` and `pairwise_F` have been plotted for $2^0, 2^1, \cdots, 2^{12}$ byte blocks. `standard_F`,`standard_U` and `pairwise_U` have been plotted only up to 2^7 bytes, since they are completely uncompetitive for large block sizes. The labels in Figures 10 and 11 indicate the implementations that make up the hull of optimality (i.e. the best implementation for a range of block sizes).

6.1 General Observations

As is to be expected, the standard exchange algorithm does well for small block sizes and the pairwise algorithm is best for large sizes. Because of the behavior of UNFORCED messages (Figure 2), there is always a drastic jump in the plots for `standard_U` and `pairwise_U`. This jump occurs at 100 bytes for `pairwise_U` but, since our plots are for block sizes that are powers of 2, we observe this jump between 2^6 and 2^7 bytes. There are also jumps in the `standard_U` plots; these occur at $(100/2^{d-1})$ bytes since the standard exchange algorithm uses messages of size 2^{d-1} blocks for d dimensional hypercubes. On our plots we observe jumps between block sizes of 2^{7-d} and

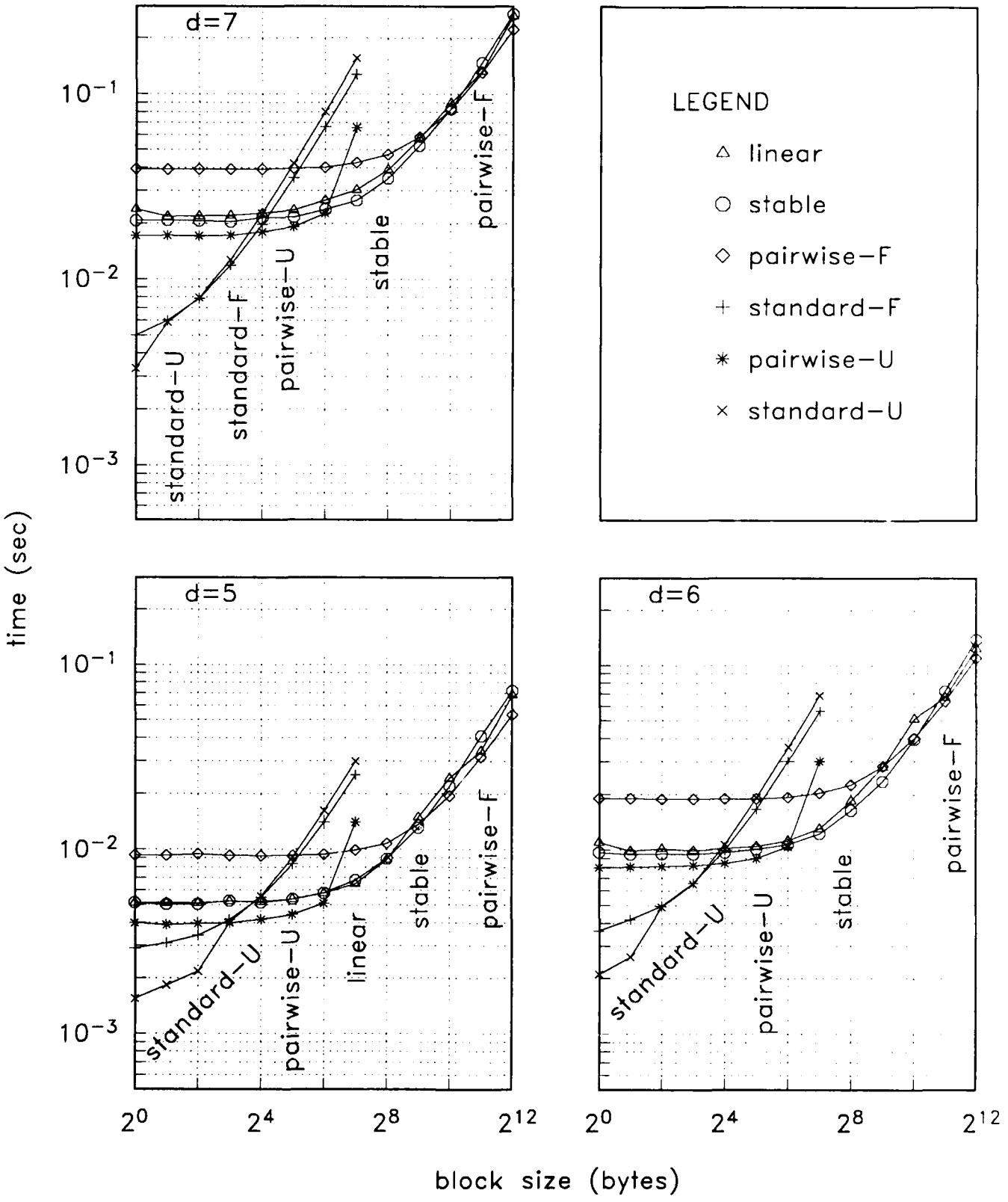


Figure 10: Comparison of implementations. $d = 5, 6, 7$

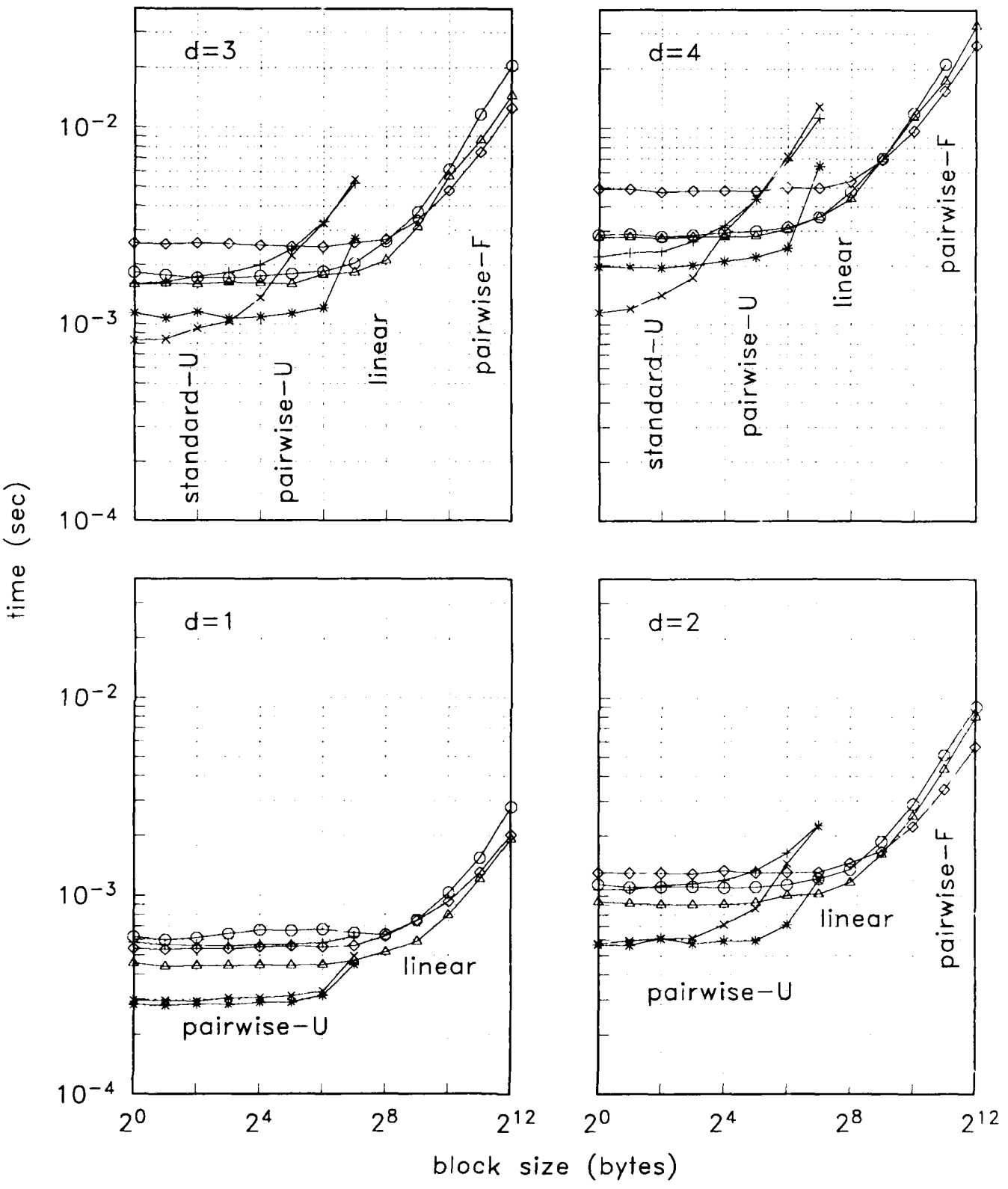


Figure 11: Comparison of implementations. $d = 1, 2, 3, 4$

2^{8-d} bytes.

6.2 Standard Exchange

standard_U is always better than standard_F for small block sizes. This is because standard_F incurs the overhead of global and pairwise synchronization, whereas standard_U does not use any form of synchronization. However there is ultimately a crossover because of the discontinuity in transmission times of UNFORCED messages discussed above. In terms of overall optimality, standard_F is useful only over a small range of message sizes for dimension 7. It is possible that this variant would be of use over a wider range on larger hypercubes[¶].

6.3 Stable and Linear

The n step stable algorithm is better than the $n-1$ step linear algorithm for large ranges of message sizes in cubes of dimension $5 \cdots 7$, demonstrating that the stability property is more useful than the number of steps (recall that these plots are of *maximum* execution time). This phenomenon is clear in Figure 12, which shows the envelopes of the linear, stable and pairwise-F implementations. The upper (lower) plot in each envelope indicates the maximum (minimum) of 1000 observations. The middle plot is the average. It can be seen that the spread of linear is slightly more than that of stable, whereas its average and maxima are well above stable.

6.4 Pairwise Algorithm

pairwise_U is about twice as fast as pairwise_F for blocks of less than 100 bytes. This is again because of the overhead of global and pairwise synchronization. Turning to Figure 12 we can see that the envelope for pairwise_F is very tight. This demonstrates how useful the pairwise synchronization procedure is in ensuring concurrent transmit/receive. The large variations for linear and stable are caused by fortuitous concurrency during some runs and serial transmit/receives during others.

[¶]The largest iPSC-860 currently available is $d = 7$.

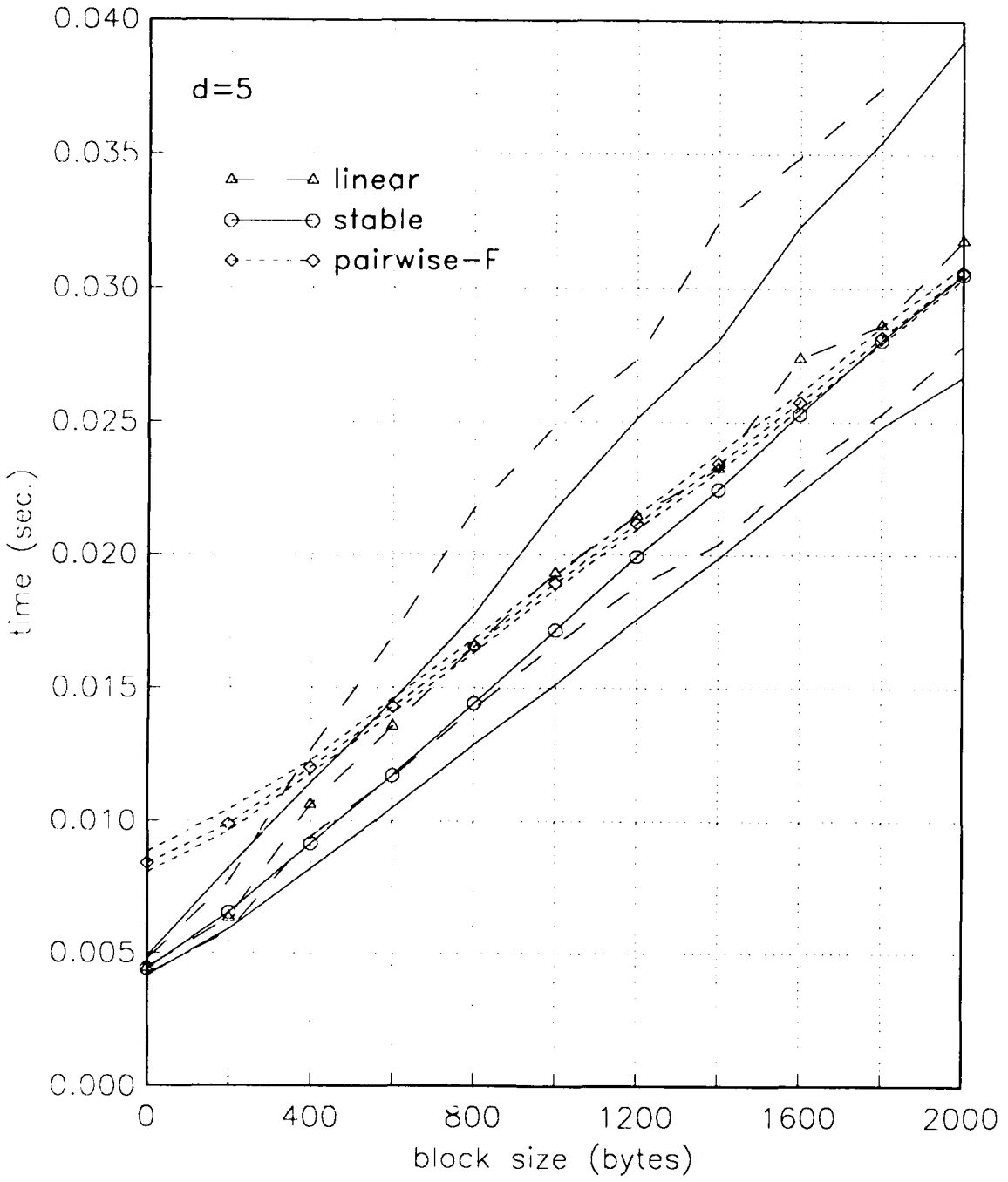


Figure 12: Detailed comparison of linear, stable and pairwise_F on a dimension 5 iPSC-860. Each set of 3 curves represents the maximum, average and minimum of 100 observations.

To explore the impact of pairwise synchronization further, we show in Figure 13 what happens to `pairwise_F` when the pairwise synchronizations are removed. The plots in the envelopes represent maximum, average and minimum values, just as in Figure 12. It can be seen that with pairwise synchronization, all timings fall within a very narrow band of constant width. Without synchronization, the difference between maximum and minimum values grows with block size. However the minimum times for the unsynchronized variant are generally better than `pairwise_F` by a constant amount because the overhead of pairwise synchronization is not being incurred. This is due to chance pairwise synchronization of all transmissions. This is a very rare event that does not occur for some block sizes in Figure 13, even though we have run our experiment for 1000 iterations each at block sizes 0, 1000, 2000, $\cdots$.

6.5 Performance on Small Hypercubes

It is interesting to study the evolution of the hull of optimality as we move from dimension 1 to 3. For dimension 1, `pairwise_U` and `linear` dominate small and large messages respectively. `pairwise_F` is never optimal for this dimension because pairwise synchronization is achieved by the global synchronizations used by `linear`, since each processor sends out only one message. The extra overhead of pairwise synchronization in `pairwise_F` is redundant. `pairwise_U` and `standard_U` degenerate into the same algorithm at this dimension, except that `standard_U` has the overhead of permutation.

As we move to dimension 2, the overhead of pairwise synchronization in `pairwise_F` pays off and it becomes optimal for large messages. It remains asymptotically optimal for all dimensions beyond 2. At dimension 3, the overhead of data permutation in `standard_U` becomes useful, and it becomes optimal for the smallest messages: this variant is optimal for small messages for all dimensions beyond 3.

6.6 Regions of Optimality

The following chart summarizes the above discussion by showing the block size and dimension for which each implementation is best.

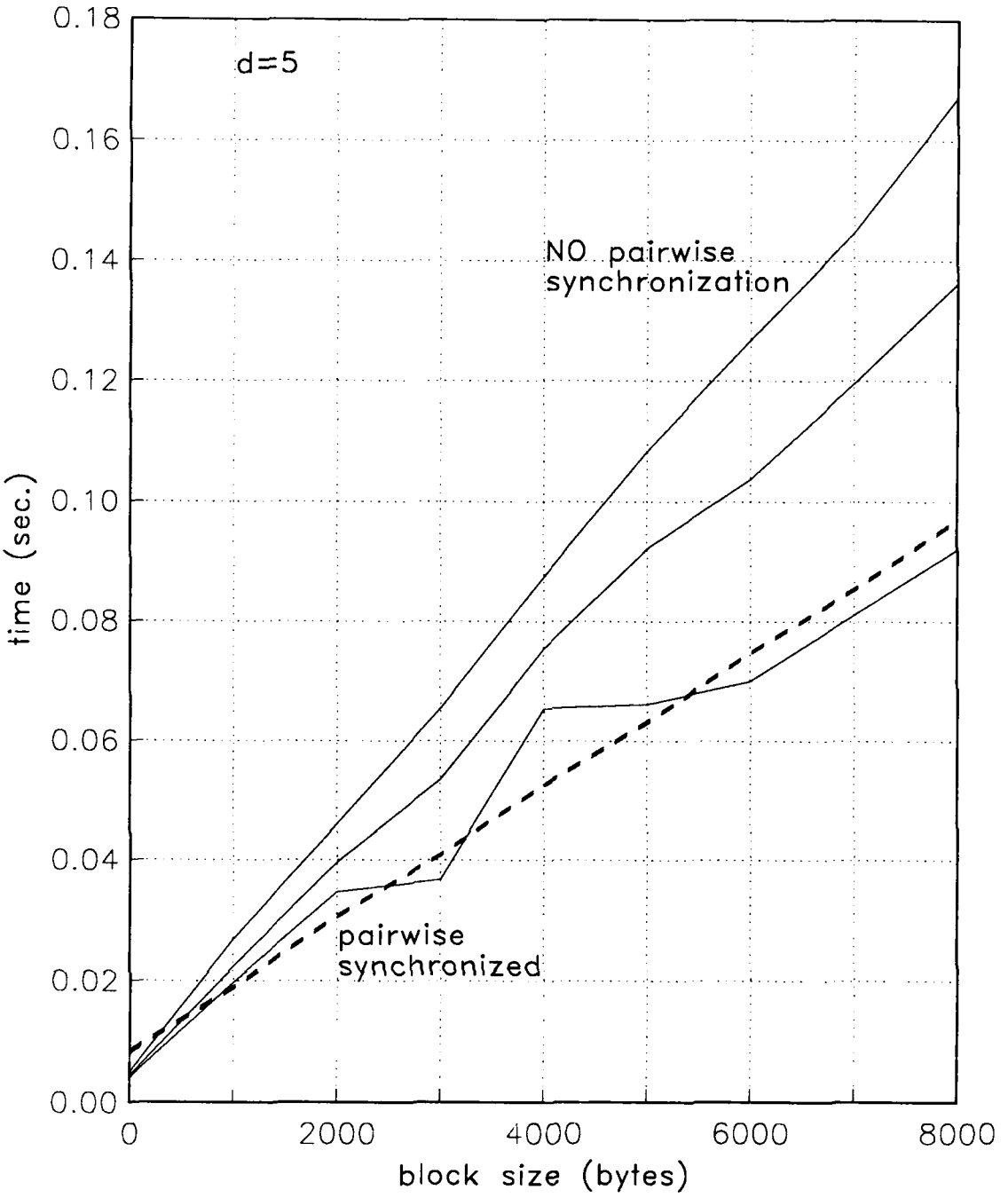
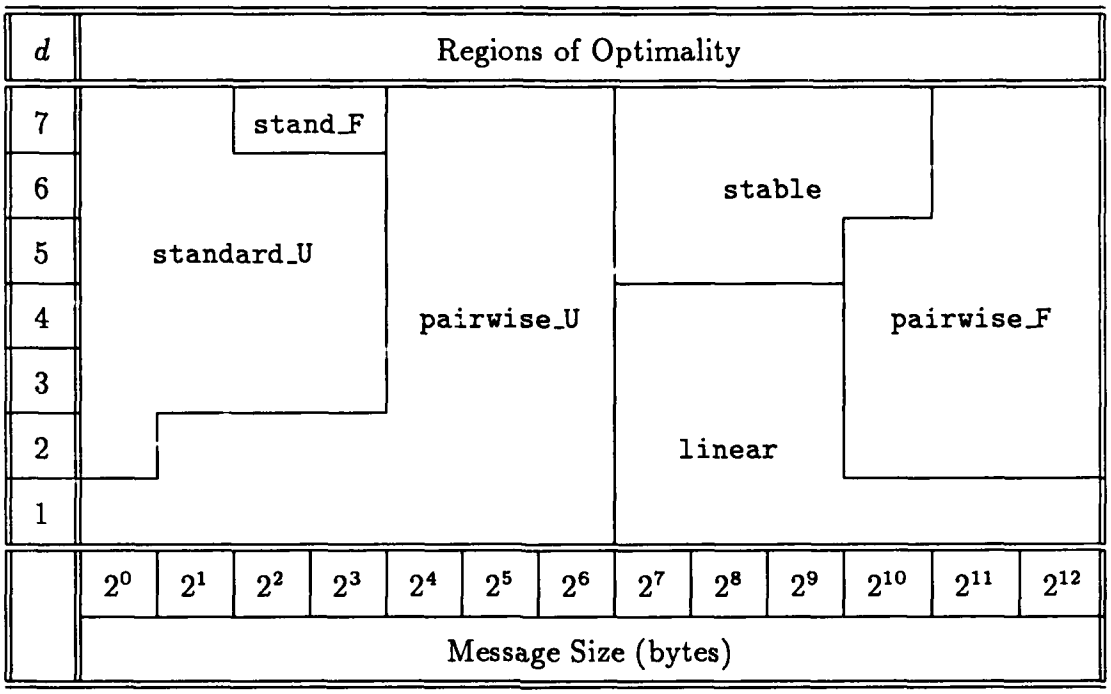


Figure 13: The impact of removing pairwise synchronization from pairwise_F on a dimension 5 iPSC-860. Each set of 3 curves represents the maximum, average and minimum of 1000 observations.



7 Conclusions

We have discussed the communication performance of the Intel iPSC-860 circuit-switched hypercube and discussed the implementation of the complete exchange on this machine. Our experimental observations show that avoidance of contention is a major consideration when scheduling communication on this machine. We have presented six implementations of complete exchange and have shown that each is useful for some range of message size and cubc dimension. Our observations of the performance of these implementations show the importance of pairwise synchronization on this machine. While the overhead of pairwise synchronization is negligible for large messages, it cannot be ignored for small messages. In the latter case, implementations without such synchronization offer considerable advantage (but are applicable only to small messages).

Since the complete exchange is a superset of any arbitrary communication pattern, the techniques and observations of this paper have broad applicability. In particular, we have found the tabular scheme introduced in Figure

5 to be very useful in planning communications. Scheduling transmission on hypercubes using 'e-cube' routing is equivalent to a numbering of the x's on this table under the constraints stated in Section 3. This scheme can easily be modified to account for any fixed routing strategy other than the widely used 'e-cube'. The techniques of this paper are certainly applicable to one-to-all broadcast, all-to-all broadcast and one-to-all personalized communications[5].

The six implementations we have described fall into two classes. These are (1) the standard exchange algorithms that are $O(n \log n)$ and are useful for small message sizes, and (2) the optimal or near-optimal $O(n)$ algorithms that always perform well for large messages. In a companion paper [2] we describe a unified multiphase algorithm that combines both classes into one. It is shown that `standard_F` and `pairwise_F` can be unified into one algorithm that outperforms either of its constituents over some ranges of message size and cube dimension. The results of the present paper can be combined with the results in [2] to obtain even faster algorithms.

In conclusion we can state that due consideration of network topology, routing strategy, message protocol and synchronization technique is necessary in order to obtain maximum performance from distributed memory multicomputers like the iPSC-860. A careless implementation can take 2 to 3 times longer than a carefully thought out schedule. A little attention to the results of this paper has the potential of improving performance by a factor of 3 or more without any major changes in code.

Acknowledgements

I wish to thank Tom Crockett for his help with `bluecrab`, the 32 node iPSC-860 at ICASE and for numerous useful discussions. Leigh Ann Tanner gave me generous assistance with `lagrange`, the 128 node iPSC-860 at NASA Ames Research Center.

References

[1] S. H. Bokhari. Communication overheads on the Intel iPSC-860 hypercube. ICASE Interim Report 10, May 1990.

[2] S. H. Bokhari. Multiphase complete exchange on a circuit switched hypercube. Technical Report 91-5, ICASE, January 1991.

[3] Intel Corporation. iPSC/2 and iPSC/860 programmers reference manual, June 1990.

[4] S. Lennart Johnsson and Ching-Tien Ho. Matrix transposition on boolean n-cube configured ensemble architectures. *SIAM J. Matrix Anal. Appl.*, 9(3):419–454, July 1988.

[5] S. Lennart Johnsson and Ching-Tien Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Computers*, C-38(9):1249–1268, September 1989.

[6] Ming-Horng Lee and Steve R. Seidel. Concurrent communication on the Intel iPSC/2. Technical Report CS-TR 9003, Dept. of Computer Science, Michigan Tech. Univ., July 1990.

[7] Thomas Schmiermund and Steve R. Seidel. A communication model for the Intel iPSC/2. Technical Report CS-TR 9002, Dept. of Computer Science, Michigan Tech. Univ., April 1990.

[8] Steve Seidel, Ming-Horng Lee, and Shivi Fotedar. Concurrent bidirectional communication on the Intel iPSC/860 and iPSC/2. Technical Report CS-TR 9006, Dept. of Computer Science, Michigan Tech. Univ., November 1990.