Communication Timing on the NCUBE Multiprocessor

David W. Krumme *
Marc Greenfield †

Technical Report 88-1
February 1988

Abstract

A series of controlled experiments is used to measure the computational and elapsed times for communication events on the NCUBE hypercube multiprocessor. Communication timing is found to be describeable by 6 parameters which depend linearly on message size. The timing consequences of interaction between computation and single and multiple communication events are also measured.

1 Introduction

Communication is crucial in a distributed memory multiprocessor, yet little is known of the timing properties of communication in machines that are in use today. This paper reports timing values for the NCUBE hypercube measured in a series of controlled experiments. It is found that communication times can be accurately described by a model that has 12 parameters.

Section 2 describes the parameters we measure. Section 3 describes our methodology. The results of the experiments are presented in Section 4. Section 5 describes some further experiments that illustrate how the timings will vary when the conditions differ from those under which the measurements were taken. Section 6 contains a summary of the results and concluding remarks. The programs we used are listed in Appendices.

2 Times to be measured

We start with a model of communication transactions involving six identifiable actions of the operating system. The times that these actions take are denoted by F, S, R, A, U, and E. In general the times vary depending on various factors, but in this work most of those factors will be controlled.

The sending process makes a system call to write a message and the operating system processes the system call: it allocates a system buffer, copies the message into it, initiates communication with a neighboring processor, and returns without waiting for the transmission to complete. (Under heavy load, it may have to wait to obtain space into which to copy the message; we avoid this case.) We define time F, the time to "format" a message, as the time from just before the system call is made to the time when the message has been

*Partially supported by Office of Naval Research contract N00014-87-K-0182.
†Partially supported by NSF grant CCR-87410750.

copied and the operating system is ready to begin transmitting it.

After the transmission is initiated and the write system call has returned, the sending process is free to carry out other computations. However, its execution speed will be impacted by the overhead caused by the transmission, including direct memory access by the hardware controller and the execution of interrupt-service routines. This loss of speed amounts to a usage of computation time which we define as S, the time to "send" the message over the communication channel. (The loss of S units of computation time is equivalent to forcing the sending process to expend S units of explicit extra computation to send the message.) The measurement of S actually begins at the point where the measurement of F stops, so S includes the time used in initiating output and returning from the write system call. The measurement of S ends when all activity on the sender related to the given transmission is finished, including cleanup steps (deallocating buffer space) at the end.

The receiving processor is activated and it cooperates with the sending processor to transmit the message concurrently with underlying computation. This cooperation may involve the exchange of overhead messages as well as the subject message itself, carried out through the repeated execution of interrupt-service routines on the sending and receiving processors. (It may also involve waiting for the receiving processor to be able to allocate buffer space for the message, but we avoid this case.) Eventually the transmission is complete and the message is located in an operating system buffer at the receiving processor. The loss of computation time experienced by underlying computations on the receiving processor up to this point is a time we define as R, the time to "receive" a message.

After the received message has been stored by the operating system, an application program can receive it through a system call. We must break this into two parts because the read system call can be issued and partially processed before the message arrives. The part of a read call that can be executed before the message arrives uses an amount of time we define as A, the time to "accept" a message. It consists primarily of making the system call. The remaining part of the read system call, including the time to copy the message, deallocate buffer space, and return from the system call, is defined as U, the time to "unformat" the message.

The above five times are all *processor* or *computation* times measured within a single processor. The sixth time is an elapsed time measured across processors: it is measured beginning at the point where a message is ready to be sent from a system buffer (*i.e.* the point where the measurement of F stops on the sending processor), and it extends until the point where the message has been received into a system buffer (*i.e.* the point where the measurement of U would begin if a prior read request had already been effected).

The above descriptions implicitly assume that the writing and reading processes are located on neighboring processors. If they are more distant from each other, the message will be forwarded by the operating system through intermediate processors. The above definitions are designed so that on such an intermediate processor, F, A, and U are eliminated from consideration, while R, S, and E are the same as before. In fact, from an operational point of view the exact points where measurement of F stops, S begins, R stops, and U begins are largely determined by time comparisons between such intermediate nodes and the sending and receiving nodes.

The actions corresponding to R, S, and E are actually *slightly* different for a forwarded message. On a sending node, S includes a re-

turn from the write system call, whereas on a forwarding node it includes a return from an interrupt. We believe the times of these steps to be comparable and small. On a receiving node, R and E include adding a new buffer to a list of pending input buffers, whereas on a forwarding node they include a routing calculation. Again we feel that these times are comparable and small.

3 Method of measurement

The six times we wish to measure vary depending on message size, the nature of underlying computations, the amount of communication activity on other I/O channels and on other nodes, and other factors. Our approach is to fix most of these factors so as to obtain reliable estimates of the values under *known* conditions. The main factor we allow to vary is the message size, since one of our main motives is to see how the values depend on message size. In general, we choose conditions relevant to the *best* or most efficient use of the multiprocessor, so that our resultant figures reflect what is realistically *possible* for a well-organized, well-balanced computation. Imbalanced or inefficient computations would generally achieve worse times for which these times can be used as a yardstick.

3.1 Conditions under which measurements are made

Our measurements are all made by a single program running under the Vertex operating system on the nodes of a 64-node NCUBE/seven with a 7 MHz clock. This program is listed in its entirety in Appendix A. The times are measured at node 0 which sends the raw values to the NCUBE host for analysis by a program running concurrently on the host. This program is listed in its entirety in Appendix B.

Measurements are made with all processors active all the time; idle processors are allowed only in cases where no timing measurements are involved and where processor activity could not possibly interact with any measurement-related activity. There are several reasons for this. First, a well-organized computation should in principle be able to achieve 100% processor utilization through various load-balancing techniques, and in many specific cases 100% utilization will be quite feasible. Second, it is technically easier in an experiment to cause a processor to remain compute-bound for say one millisecond than to cause it to be idle for one millisecond. And third, we find it more interesting to see how fast communication might be on a busy system than how fast it might be on an inactive one.

We perform communication one message at a time, never allowing different I/O channels on a processor to be active at the same time. Thus whenever communication hardware is active, it contends the central processor for memory access, and with no one else.

Our experience has indicated that on the Intel hypercube multiprocessor, the nature of the instructions being executed by the processor can affect the memory contention between the processor and an I/O controller, so we arrange that a particular chosen loop will be in execution whenever communication that we are timing is in progress. We shall call this chosen underlying computation Q. We ran experiments with two different versions of Q: one involving integer arithmetic (adds, multiplies, compares) and one involving floating point arithmetic (add and multiplies plus integer loop control instructions). For the NCUBE hardware, the actual transmission rate is determined solely by the sender (there is no mechanism for the receiving controller to tell the sending controller to slow

down), so message transmission speed can be affected by processor activity on the sending but not on the receiving node.

3.2 Measurement of E

We measure E through a simple echo test. Node 0 sends to node N which, after some fixed computation that takes time X (giving node 0 time to prepare for the return message), sends the message back to node 0. After all write system calls, the nodes perform computation Q for a sufficiently long time that the output can complete (so that output is done with computation active). The time measured at node 0 from before the send to after the read should be $2F + 2dE + 2U + X$, where d is the distance (number of hops) the message travels in going from 0 to N. We perform this test for $d = \{1, 2, 3, 4, 5\}$, obtaining times T_d representing the average times for the echo test at distance d. We can then calculate E as $\frac{1}{2}(T_{d+1} - T_d)$ for $d = \{1, 2, 3, 4\}$.

3.3 Measurement of R

For R, we have node 0 send a message to a neighbor which computes for a known fixed amount of time X and then sends the message back. Node 0 performs Q long enough for the output to clear, then it carries out a long timed computation of Q during which the echoed message will be received by the operating system. Finally it reads the message, just to clear it out. The excess time taken for the long computation of Q, over the time measured with no I/O in progress, is a direct measure of R.

3.4 Measurement of S, F, U, and A

The other values are calculated by obtaining times from four different experiments and solving the resulting system of four equations in four unknowns. The first experiment simply arranges to have an unread message in the possession of the operating system and then making a timed read system call for it. This time should be A + U.

The second experiment simply times a write system call followed by a fixed computation Q, subtracting the known time the computation of Q should take without I/O in progress. This value should be F + S.

The third experiment is similar to the experiment that measures R. Node 0 sends a message to a neighbor, performs Q long enough for the output to clear, and then carries out a long timed computation of Q. The neighbor echoes the message in such a way that it passes through node 0 during the computation of Q and is forwarded by the operating system to another neighbor. The extra time taken by Q should then be R + S.

In the fourth experiment, node 0 sends a message to a neighbor, performs Q long enough for the output to clear, and then immediately issues a read system call before the message has been echoed by the neighbor. The neighbor, which issued a read system call well in advance of the receipt of the message, performs a known fixed computation for time X after receiving the message and then echoes it back. The time measured on node 0 from before the write to after the read should be $F + E + U + X + F + E + U$, so that after subtracting X one has a measurement of $2(F + E + U)$.

3.5 Calculations

The parameters we are measuring in these experiments are not amenable to highly precise measurement, nor is great precision an important goal. If we can determine the parameters to within 50 microseconds, and be confident in those values, then that will be sufficient.

We repeat each test 100 times and record mean values for the timed quantities. The times reported by the system have a granularity of about 140 microseconds, so this gives us raw data that should be accurate to just a few microseconds.

Timing measurements yield for each message size n values for $E, R, R+S, F+S, F+E+U$, and $A+U$. We simply proceed through this list left-to-right and at each step we find just one new unknown, so we can calculate the values E, R, S, F, U, and A in order. Since each of the last three steps uses values produced in the preceeding step, we can expect decreasing accuracy caused by the accumulation of errors.

The calculated values generally prove to depend linearly on n, and we fit the observed points with lines defined by $E_0 + E_1 n$, $R_0 + R_1 n$, $S_0 + S_1 n$, $F_0 + F_1 n$, $U_0 + U_1 n$, and $A_0 + A_1 n$. We use the least-squares method to fit these lines.

4 Results

Table 1 presents the calculated values for E, R, S, F, U, and A for 16-, 500-, 1000-, 1500-, and 2000-byte messages, where the underlying computation Q used integer arithmetic. The experiment was repeated five times, and the values shown are means of the five. The standard deviation is shown below each value.

Table 2 presents the same data for the case where the underlying computation Q used floating point arithmetic. No significant difference is apparent between the floating point and integer versions.

The results from Table 1 are displayed graphically in Figure 1. The points can be fit nicely by line segments, and Table 3 shows the result of fitting linear equations to the timing values, where the resulting values have been expressed with a number of significant digits that seems appropriate.

Size	16	500	1000	1500	2000
E	242	864	1506	2149	2792
	± 1	± 2	± 1	± 1	± 0
R	178	200	219	249	261
	± 3	± 2	± 2	± 1	± 2
S	152	157	167	167	175
	± 3	± 2	± 2	± 2	± 2
F	169	465	762	1036	1372
	± 3	± 3	± 3	±18	± 3
U	137	429	748	1072	1343
	± 2	± 4	± 4	±18	± 3
A	122	124	111	97	126
	± 5	± 4	± 3	±18	± 2

Table 1: Times in microseconds measured with integer arithmetic underlying communication. Means and standard deviations for five trials are shown.

Size	16	500	1000	1500	2000
E	245	868	1509	2154	2797
	± 1	± 1	± 0	± 1	± 1
R	176	202	229	250	275
	± 3	± 9	± 5	± 2	± 3
S	131	156	162	166	169
	±23	± 7	± 7	± 2	± 5
F	193	463	767	1071	1377
	±22	± 5	± 9	± 4	± 4
U	112	428	734	1036	1335
	±21	± 9	±11	± 5	± 6
A	148	126	123	128	129
	±21	± 8	± 9	± 2	± 6

Table 2: Times in microseconds measured with floating point arithmetic underlying communication. Means and standard deviations for five trials are shown.

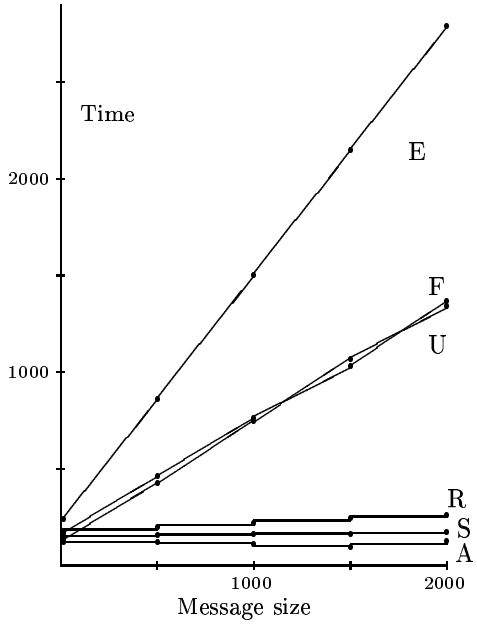

Figure 1: Data from Table 1.

E	$220 + 1.285n$
R	$180 + .04n$
S	$150 + .01n$
F	$160 + .60n$
U	$130 + .61n$
A	$120 + 0.0n$

Table 3: Linear formulas expressing the six times as functions of the message size n.

5 Other measurements

Two further experiments were conducted to find out how the values behave when the conditions vary. We are interested in whether communication is faster when there is no underlying computation and in what happens when several channels are active at once.

5.1 Effect of computation on communication

The measurement of E (Section 3.2) was repeated with all processors idle instead of computing. This yielded an improvement of one to four microseconds, an insignificant amount. Hence communication latency (with one channel active) depends not upon whether the processor is idle or busy computationally.

5.2 Interaction of communication activities

To see the effect of having several communication channels active simultaneously, we ran the following experiment. We let one node measure the time taken by a fixed computation while one, two, or three neighbors send 2000-byte messages continuously that get forwarded through it. Each neighbor's messages will cause two channels to be active as the forwarding of one message overlaps the receipt of the next one. Thus this experiment measures the effect of two, four, and six channels being active.

First we check the rate at which the messages move. With 2, 4, and 6 channels active, transfers from each neighbor occur at an average rate of 2800, 2830, and 3070 microseconds per message respectively. With actual transmission the limiting factor, based on the data calculated with one channel active one would predict E(2000) or 2790 microseconds per message. Clearly interference among channels be-

gins to be significant with 6 channels active. With 4 or fewer channels active, messages do not impede each other significantly.

The observed effect at the node timing the underlying computation was this: if the time taken by the computation with no I/O in progress was 1.0, then the time taken with 2, 4, and 6 channels active was 1.20, 1.61, and 3.78 respectively. To interpret these figures the execution of interrupt-service routines must be accounted for. For each neighbor sending messages, a computational load is experienced in the amount of $R(2000) + S(2000)$ each time a message passes through node 0. This means approximately 440, 880, or 1320 microseconds every 2800, 2830, or 3070 microseconds. Thus in each second, only .84, .69, and .57 seconds respectively will be available for the computation, so the computation should take 1.19, 1.45, and 1.75 units of time respectively if there are no interaction effects. Comparison with the observed figures of 1.20, 1.61, and 3.78 indicates growing interference which jumps from a minimal level to a dramatic level in going from 4 to 6 channels.

6 Discussion

The figures in Table 3 are easily interpreted. Only E, F, and U depend significantly on message size. For E, this reflects the rate at which the hardware transfers data, while for F and U it represents the copying of data by the operating system between user and system buffers. The fixed components of the times largely represent software steps: for A, F, and U this includes system entry and return and for R and S it includes the execution of interrupt-service routines. The fixed time for E includes an amount representing a request/reply sequence exchanged initially between sending and receiving nodes.

Using these results to predict the behavior of complete programs requires care. For a single message, the points at which R and S are incurred are unpredictable, and furthermore E can overlap R, S, and A. For a sequence of messages, the asynchronous relationships among the processors means that there is much opportunity for sequencing relationships to develop that confound analysis. For example E can be incurred concurrently with all the other values, causing the overall elapsed times to improve considerably. (We have commonly observed this effect.)

One interesing observation is that for unbuffered communication of large messages, it is possible to move large amounts of data at the maximum rate with only a small cost in compute time. This is because if buffer copying is eliminated, both the sending cost $F + S$ and the receiving cost $A + U + R$ will be less than 400 microseconds, as compared with an elapsed transmission time of almost 3 milliseconds for a 2000-byte message.

Deadlock is an important problem in message-passing systems, and these timing figures indicate that a natural form of deadlock prevention is occuring: the time taken for a user process to send a message is between 50% and 100% of the time it takes to transmit it, and the time needed for a user process to forward a message exceeds the time required to transmit it. This means that programs are generally not able to deliver messages to the communication system at a rate that greatly exceeds its ability to deliver them, thus limiting the possibilities for deadlock.

A Measurement program

```
/*
 * Communication performace timing program for the Ncube
 */

#include <local/vertex.h>
#define QFUNC qint
#define NOTIMES 100
#define COMPTIME 25
#define SENDTIME 5
#define QINTLOOP 40
#define QFLOATLOOP 18
#define CHECKCOUNT 60000
#define BIGSIZE 2000
#define PARTSIZE 2

#define SEND 0
#define COMP 1
#define NTIM 2

#define FNOTIMES ((double)NOTIMES)

int node, procid, host, dim, status;
int solutions[NOTIMES];
char buffer[2050];
int sizes[5] = {16, 500, 1000, 1500, 2000};
float sol[5][5][3];
float times[3];
float check;
int icheck;
float log[NOTIMES];

main() {
    int src, type, a, i, s;
    int tbase, start, finish, finish2, finish3;
    int pass2, dst;
    float time, min, max, sum;
    float comptime, sendtime;

    whoami(&node, &procid, &host, &dim);
    type = 0x100;
    pass2 = 0;
    /**** carry out dimension exchange to get in sync ***/
    if (node < 32) for (i=0; i<5; ++i) {
        src = node ^ (1<<i);
        if (node < src) {
                nwrite(&check, 4, src, type, &status);
                nread(&check, 4, &src, &type, &status);
        } else {
                nread(&check, 4, &src, &type, &status);
                nwrite(&check, 4, src, type, &status);
        }
    }
```

```
    switch(node) {
    case 0:
        /**** get known compute values ****/
        i = NOTIMES;
        tbase = ntime();
        start = ntime()-tbase;
        do { ntime();ntime();ntime();ntime();ntime();ntime();
             ntime();ntime();ntime();ntime();ntime();ntime();
        } while (--i > 0);
        finish = ntime()-tbase;
        times[NTIM] = (double)(finish - start) / (FNOTIMES*12 + 1);
        i = 10;
        start = ntime()-tbase;
        do { QFUNC(COMPTIME); QFUNC(COMPTIME);
             QFUNC(COMPTIME); QFUNC(COMPTIME);
             QFUNC(COMPTIME); QFUNC(COMPTIME);
             QFUNC(COMPTIME); QFUNC(COMPTIME);
             QFUNC(COMPTIME); QFUNC(COMPTIME);
        } while (--i > 0);
        finish = ntime()-tbase;
        comptime = times[COMP] = (double)(finish - start - times[NTIM])/100.;
        sendtime = times[SEND] = comptime * SENDTIME / COMPTIME;

        /**** send times to host ****/
        nwrite(times, 12, host, 0x200, &status);
            QFUNC(SENDTIME);

repeat0: /**** test 1: E ****/
        for (i=1;i<6;i++) {
            for (s=0;s<5;s++) {
                sum = max = 0;
                min = 999;
                for(a=0;a<NOTIMES+1;a++) {
                    src = (1 << i) - 1;
                    start = ntime()-tbase;
                    nwrite(buffer, sizes[s], src, type, &status);
                    if (!pass2) QFUNC(SENDTIME);
                    nread(buffer, sizes[s], &src, &type, &status);
                    finish = ntime()-tbase;
                    time = finish - start - sendtime - times[NTIM];
                    if (a > 0) {
                        if (time < min) min = time;
                        if (time > max) max = time;
                        sum += time;
                    }
                    QFUNC(SENDTIME);
                }
                sol[s][i-1][0] = (sum/FNOTIMES)-times[NTIM];
                sol[s][i-1][1] = min;
                sol[s][i-1][2] = max;
            }
        }
        nwrite(sol, 75*4, host, 0x200, &status);
        if (pass2) goto final;
```

```
/**** clear sol var for re-use ****/
for (a=0;a<5;a++)
    for (s=0;s<5;s++) {
        sol[a][s][0] = 0.0;
        sol[a][s][1] = 99.0;
        sol[a][s][2] = 0.0;
    }

/**** check that all nodes ran long enough ****/
src = 4; /* everyone else runs longer than 4 */
start = ntime()-tbase;
nread(&check, 4, &src, &type, &status);
finish = ntime()-tbase;
check = finish - start;
nwrite(&check, 4, host, 0x200, &status);
/**** wait for all nodes to end compute loops ***/
src = 31; /* this will be the last to finish */
nread(&check, 4, &src, &type, &status);

/**** test 2-4: R, A+U, F+S ****/
src = 2;
for (s=0;s<5;s++) {
    for (a=0;a<NOTIMES+1;a++) {
        nwrite(buffer, sizes[s], 2, type, &status);
        QFUNC(SENDTIME);
         start = ntime()-tbase;
        QFUNC(COMPTIME);
         finish = ntime()-tbase;
        nread(buffer, sizes[s], &src, &type, &status);
         finish2 = ntime()-tbase;
        nwrite(buffer, sizes[s], 2, type, &status);
        QFUNC(SENDTIME);
         finish3 = ntime()-tbase;
        nread(buffer, sizes[s], &src, &type, &status);

        if (a>0) {
            time = finish - start - comptime;
            if (time < sol[s][0][1]) sol[s][0][1] = time;
            if (time > sol[s][0][2]) sol[s][0][2] = time;
            sol[s][0][0] += time;

            time = finish2 - finish;
            if (time < sol[s][1][1]) sol[s][1][1] = time;
            if (time > sol[s][1][2]) sol[s][1][2] = time;
            sol[s][1][0] += time;

            time = finish3 - finish2 - sendtime;
            if (time < sol[s][2][1]) sol[s][2][1] = time;
            if (time > sol[s][2][2]) sol[s][2][2] = time;
            sol[s][2][0] += time;
        }
        QFUNC(SENDTIME);
    }
    sol[s][0][0] /= NOTIMES; sol[s][0][0] -= times[NTIM];
    sol[s][1][0] /= NOTIMES; sol[s][1][0] -= times[NTIM];
```

```
            sol[s][2][0] /= NOTIMES; sol[s][2][0] -= times[NTIM];
        }
        nwrite(buffer, 2, host, 0x200, &status);

        /**** test 5: R+S ****/
        src = 1;
        nwrite(buffer, 2, 1, type, &status);
        nread(buffer, 2, &src, &type, &status);
        for (s=0;s<5;s++) {
            for (a=0;a<NOTIMES+1;a++) {
                nwrite(buffer, sizes[s], 1, type, &status);
                QFUNC(SENDTIME);
                start = ntime()-tbase;
                QFUNC(COMPTIME);
                finish = ntime()-tbase;
                time = finish - start - comptime;
                if (a>0) {
                    if (time < sol[s][3][1]) sol[s][3][1] = time;
                    if (time > sol[s][3][2]) sol[s][3][2] = time;
                    sol[s][3][0] += time;
                }
                QFUNC(SENDTIME);
            }
            sol[s][3][0] /= NOTIMES; sol[s][3][0] -= times[NTIM];
        }
        nwrite(buffer, 2, host, 0x200, &status);

        /**** test 6: F+E+U ****/
        src = 2;
        for (s=0;s<5;s++) {
            for (a=0;a<NOTIMES+1;a++) {
                start = ntime()-tbase;
                nwrite(buffer, sizes[s], 2, type, &status);
                QFUNC(SENDTIME);
                nread(buffer, sizes[s], &src, &type, &status);
                finish = ntime()-tbase;
                time = finish - start - comptime;
                if (a>0) {
                    if (time < sol[s][4][1]) sol[s][4][1] = time;
                    if (time > sol[s][4][2]) sol[s][4][2] = time;
                    sol[s][4][0] += time;
                }
                QFUNC(SENDTIME);
            }
            sol[s][4][0] /= NOTIMES; sol[s][4][0] -= times[NTIM];
        }
        nwrite(sol, 75*4, host, 0x200, &status);

        /**** test 7:  redo test 1 w/o computation ****/
        QFUNC(COMPTIME); /* give a little space */
        pass2 = 1;
        goto repeat0;

final:  /**** test 8:  R+S continuous X 1,2,3 ****/
        for (i=0; i<3; ++i) {
```

```
            nwrite(buffer, 4, 1<<(i+i), type, &status);
            QFUNC(SENDTIME);
            a = 10;
            start = ntime()-tbase;
            do { QFUNC(COMPTIME); QFUNC(COMPTIME);
                 QFUNC(COMPTIME); QFUNC(COMPTIME);
                 QFUNC(COMPTIME); QFUNC(COMPTIME);
                 QFUNC(COMPTIME); QFUNC(COMPTIME);
                 QFUNC(COMPTIME); QFUNC(COMPTIME);
            } while (--a > 0);
            finish = ntime()-tbase;
            sol[0][0][i] = (finish - start - times[NTIM])/100.;
        }
        for (i=0; i<3; ++i) {
            nwrite(buffer, 4, src=1<<(i+i), type, &status);
            nread(&sol[0][1][i], 4, &src, &type, &status);
            nread(&sol[0][2][i], 4, &src, &type, &status);
        }
        nwrite(sol, sizeof(sol[0]), host, 0x200, &status);
        break;

    case  1:
    case  3:
    case  7:
    case 15:
    case 31:

repeat1: /**** test 1 ****/
        for (s=0;s<5;s++) {
            src = 0;
            for (a=0;a<NOTIMES+1;a++) {
                nread(buffer, sizes[s], &src, &type, &status);
                QFUNC(SENDTIME);
                nwrite(buffer, sizes[s], 0, type, &status);
                if (!pass2) QFUNC(SENDTIME);
            }
        }
        if (pass2) {
            if (node == 1) goto send;
            break;
        }
        pass2 = 1;
        QFUNC(CHECKCOUNT+(CHECKCOUNT>>2));
        if (node == 31) nwrite(&check, 4, 0, type, &status);
        if (node != 1) goto repeat1;

        /**** test 5 ****/
        nread(buffer, 2, &src, &type, &status);
        nwrite(buffer, 2, src, type, &status);
        for (s=0;s<5;s++) {
            src = 0;
            for (a=0;a<NOTIMES+1;a++) {
                nread(buffer, sizes[s], &src, &type, &status);
                QFUNC(SENDTIME);
                QFUNC(SENDTIME); /* make sure return in COMP */
```

```
                nwrite(buffer, sizes[s], 2, type, &status);
                QFUNC(SENDTIME);
            }
        }
        goto repeat1;

    case  2:
        QFUNC(CHECKCOUNT+(CHECKCOUNT>>2));

        /**** test 2-4 ****/
        for (s=0;s<5;s++) {
            src = 0;
            for (a=0;a<NOTIMES+1;a++) {
                nread(buffer, sizes[s], &src, &type, &status);
                QFUNC(SENDTIME);
                QFUNC(SENDTIME); /* make sure return in COMP */
                nwrite(buffer, sizes[s], 0, type, &status);
                QFUNC(SENDTIME);
                nread(buffer, sizes[s], &src, &type, &status);
                QFUNC(SENDTIME);
                QFUNC(SENDTIME); /* make sure timing finished */
                nwrite(buffer, sizes[s], 0, type, &status);
            }
        }

        /**** test 5 ****/
        for (s=0;s<5;s++) {
            src = 1;
            for (a=0;a<NOTIMES+1;a++)
                nread(buffer, sizes[s], &src, &type, &status);
        }

        /**** test 6 ****/
        for (s=0;s<5;s++) {
            src = 0;
            for (a=0;a<NOTIMES+1;a++) {
                nread(buffer, sizes[s], &src, &type, &status);
                QFUNC(COMPTIME);
                nwrite(buffer, sizes[s], 0, type, &status);
                QFUNC(SENDTIME);
            }
        }
        goto recv;

    case 4:
        QFUNC(CHECKCOUNT);
        nwrite(&check, 4, 0, type, &status);
        goto send;
    case 16:
        QFUNC(CHECKCOUNT+(CHECKCOUNT>>2));

send:   /**** test8 ****/
        dst = node << 1; /* 1 <-> 2, 4 <-> 8, 16 <-> 32 */
        src = 0;
        nread(&check, 4, &src, &type, &status);
```

```
        for (i=0;i<3;++i)
            nwrite(buffer, BIGSIZE, dst, type, &status);
        pass2 = 0;
        a = 0; do {
         nwrite(buffer, BIGSIZE, dst, type, &status);
            src = -1;
            nread(&icheck, 4, &src, &type, &status);
            if (++a == 100) {
                if (!pass2) start = ntime();
                else {finish2 = finish3; finish3 = ntime(); a = 0;}
            }
            if (a == 200) {
                finish = ntime();
                finish3 = finish;
                pass2 = 1; a = 0;
            }
        } while (src == dst);
        check = (finish - start) / 100.;
        nwrite(&check, 4, 0, type, &status);
        check = (finish3 - finish2) / 100.;
        nwrite(&check, 4, 0, type, &status);
        break;

    case 8:
    case 32:
        QFUNC(CHECKCOUNT+(CHECKCOUNT>>2));

recv:   /**** test 8 ****/
        dst = node >> 1;
        for (;;) {
            src = -1;
            nread(buffer, PARTSIZE, &src, &type, &status);
            if (src != dst) goto out;
            nwrite(&icheck, 4, dst, type, &status);
        }

    default:
        QFUNC(CHECKCOUNT+(CHECKCOUNT>>2));
        break;
    }

    /* collectively decide to exit */
    if (node == 0) {
        for (src=1; src<=32; ++src) {
            nwrite(&icheck, 4, src, type, &status);
        }
    } else {
        src = 0;
        nread(&icheck, 4, &src, &type, &status);
    }
out: /* exit */
    nwrite(&procid, 4, host, 0x500, &status);
}

int iss;
```

```
qint(rept) {
    int i, j, k;
    int n;

    while (--rept >= 0) {
        i = 2; j = 3; iss = 0;
        for (n=0; n<QINTLOOP; ++n) {
            k = i + j;
            i = j;
            j = k;
            iss += k*k;
        }
    }
}

double fss;
qfloat(rept) {
    double i, j, k;
    int n;

    while (--rept >= 0) {
        i = 2.; j = 3.; fss = 0.;
        for (n=0; n<QFLOATLOOP; ++n) {
            j += i;
            fss += j * j;
            i += j;
            fss += i * i;
        }
    }
}
```

B Analysis program

```
/*
 * Analysis of data generated by NCUBE communication performance program
 */

#include <stdio.h>
#include <local/vortex.h>

/**** ntime -> real time defines ****/
#define     MHZ 7.0
#define     USPT (1024.0 / MHZ)

/**** times defines ****/
#define     SEND 0
#define     COMP 1
#define     NTIM 2

/**** maxdiff defines ****/
#define     RDATA 0
#define     AUDATA 1
#define     FSDATA 2
#define     RSDATA 3
#define     EDATA 5

/**** structs ****/
typedef struct EQU {
    float base;
    float slope;
    float error;
};

/**** vars to receive data in ****/
float sol1[5][5][3], sol2[5][5][3];
float times[3];
float check;

/**** work vars ****/
float diff, sum, avg[5], base, pred;
int a,b,i,shrt;
int sizes[5] = {16, 500, 1000, 1500, 2000};
int pass2;
int eflag, oflag, cflag, sflag;
float maxdiff;
double data[5];
struct EQU E, R, A, F, U, S, AU, FS, RS, EFUx2, FSAUR, AUR;
struct EQU *equar[6];

main(argc,argv) int argc; char *argv[]; {
    char *ap;

    /**** process arguments ****/
    if (argc > 1) {
        ap = argv[1];
        a = 1;
```

```
        while (a) {
            switch (*ap++) {
                case 'e':
                    eflag++;
                    break;
                case 'o':
                    oflag++;
                    break;
                case 'c':
                    cflag++;
                    break;
                case 's':
                    sflag++;
                    break;
                default:
                    a = 0;
                    break;
            }
        }
    }
    pass2 = 0;

    /**** get data from the cube ****/
    n_fread(0, times, 12);
    printf("Received constant times:\n");
    printf("  qint(COMPTIME) = %f (%f usecs).\n",times[COMP],times[COMP]*USPT);
    printf("  qint(SENDTIME) = %f (%f usecs).\n",times[SEND],times[SEND]*USPT);
    printf("  ntime()        = %f (%f usecs).\n",times[NTIM],times[NTIM]*USPT);
    n_fread(0, sol1, 300);
    printf("Test1 complete: latency test.\n");
    n_fread(0, &check, 4);
    printf("Check count = %f (%f usecs).\n", check, check*USPT);
    n_fread(0, sol2, 300);
    printf("Test2 complete: receive test.\n");
    printf("Test3 complete: receive A plus unformat test.\n");
    printf("Test4 complete: format plus send test.\n");
    n_fread(0, sol2, 300);
    printf("Test5 complete: receive plus send test.\n");
    n_fread(0, sol2, 300);
    printf("Test6 complete: unformat test.\n");
    printf("Testing complete, processing data...\n");

    if (eflag) edata_print();
    if (oflag) odata_print();

repeat:
    /**** edata: get average of the diffs between 1/2 2/3 etc node interm ****/
    for (a=0;a<5;a++) {
        sum = 0;
        for (b=1;b<5;b++)
            sum += sol1[a][b][0] - sol1[a][b-1][0];
        avg[a] = (sum / 8.0) * USPT;
    }

    /**** edata: det avg per byte of data and put in E ****/
```

```
for (a=0;a<5;a++) data[a] = (double)avg[a];
fit(&E, data, 5);

if (pass2) goto final;

/**** odata: convert all sol2 data to usecs ****/
for (a=0;a<5;a++) for (i=0;i<5;i++) sol2[a][i][0] *= USPT;

/**** odata: det avg per byte of data ****/
equar[0] = &R;
equar[1] = &AU;
equar[2] = &FS;
equar[3] = &RS;
equar[4] = &EFUx2;

for (i=0;i<5;i++) {
    for (a=0;a<5;a++) data[a] = (double)sol2[a][i][0];
    fit(equar[i], data, 5);
}

if (sflag) sdata_print();
if (cflag) cdata_print();

/**** calculate indiv stats ****/

/**** S = RS - R ****/
for (i=0;i<5;i++) {
    sol2[i][3][0] -= sol2[i][0][0];
    data[i] = sol2[i][3][0];
    if (cflag) printf("S(%4d) = %f\n", sizes[i], data[i]);
}
fit(&S, data, 5);

/**** F = FS - S ****/
for (i=0;i<5;i++) {
    sol2[i][2][0] -= data[i];
    data[i] = sol2[i][2][0];
    if (cflag) printf("F(%4d) = %f\n", sizes[i], data[i]);
}
fit(&F, data, 5);

/**** U = EFU - E - F ****/
for (i=0;i<5;i++) {
    sol2[i][4][0] /= 2;
    sol2[i][4][0] -= avg[i] + data[i];
    data[i] = sol2[i][4][0];
    if (cflag) printf("U(%4d) = %f\n", sizes[i], data[i]);
}
fit(&U, data, 5);

/**** A = AU - U ****/
for (i=0;i<5;i++) {
    sol2[i][1][0] -= data[i];
    data[i] = sol2[i][1][0];
    if (cflag) printf("A(%4d) = %f\n", sizes[i], data[i]);
```

```
    }
    fit(&A, data, 5);

    /**** print final test results ****/
    printf("\n\n");
    printf("E (Transfer Latency):  %5.1f + %4.3fn usecs. [max. error %f]\n\n",
        E.base,E.slope,E.error);
    printf("F (Format + Req. Write):  %5.1f + %4.3fn usecs. [max. error %f]\n",
        F.base,F.slope,F.error);
    printf("S (Send Load):  %5.1f + %4.3fn usecs. [max. error %f]\n\n",
        S.base,S.slope,S.error);
    printf("A (Req. Read):  %5.1f + %4.3fn usecs. [max. error %f]\n",
        A.base,A.slope,A.error);
    printf("U (Unformat):   %5.1f + %4.3fn usecs. [max. error %f]\n",
        U.base,U.slope,U.error);
    printf("R (Receive Load):  %5.1f + %4.3fn usecs. [max. error %f]\n\n",
        R.base,R.slope,R.error);

    /**** now proceed to extra tests 7 and 8 ****/
    n_fread(0, sol1, 300);
    printf("\n\nTest7 complete: unloaded latency test.\n");
    if (eflag) edata_print();
    pass2 = 1;
    goto repeat;
final:
    printf("\n\n");
    printf("Transfer Latency/Byte        = %5.1f + %4.3fn usecs.\n", E.base,E.slope);
    n_fread(0, sol2, sizeof(sol2[0]));
    printf("\n\nTest 8:\n");
    printf("  qint(COMPTIME) w/ R+S(1) = %f (%f usecs).\n",
        sol2[0][0][0],sol2[0][0][0]*USPT);
    printf("  qint(COMPTIME) w/ R+S(2) = %f (%f usecs).\n",
        sol2[0][0][1],sol2[0][0][1]*USPT);
    printf("  qint(COMPTIME) w/ R+S(3) = %f (%f usecs).\n",
        sol2[0][0][2],sol2[0][0][2]*USPT);
    printf("  ticks/buffer @ R+S(1) = %f (%f usecs/buffer).\n",
        sol2[0][1][0],sol2[0][1][0]*USPT);
    printf("  ticks/buffer @ R+S(2) = %f (%f usecs/buffer).\n",
        sol2[0][1][1],sol2[0][1][1]*USPT);
    printf("  ticks/buffer @ R+S(3) = %f (%f usecs/buffer).\n",
        sol2[0][1][2],sol2[0][1][2]*USPT);
    printf("final ticks/buffer @ R+S(1) = %f (%f usecs/buffer).\n",
        sol2[0][2][0],sol2[0][2][0]*USPT);
    printf("final ticks/buffer @ R+S(2) = %f (%f usecs/buffer).\n",
        sol2[0][2][1],sol2[0][2][1]*USPT);
    printf("final ticks/buffer @ R+S(3) = %f (%f usecs/buffer).\n",
        sol2[0][2][2],sol2[0][2][2]*USPT);
}

edata_print() {
    int a, b;
    float sum, avg[5], diff;

    printf("Bufs N Averg min max\n");
    for (a=0;a<5;a++) {
```

```
        for (b=0;b<5;b++) {
            printf("%4d %1d %6.2f %6.2f %6.2f\n", sizes[b], a+1,
                sol1[b][a][0], sol1[b][a][1], sol1[b][a][2]);
        }
        printf("\n");
    }
    printf("--------------------------------\n");
    printf("Bufs N/N Diff\n");
    for (a=0;a<5;a++) {
        sum = 0;
        for (b=1;b<5;b++) {
            sum += (diff = sol1[a][b][0] - sol1[a][b-1][0]);
            printf("%4d %1d/%1d %4.2f\n", sizes[a], b-1, b, diff);
        }
        printf("\n");
        avg[a] = sum / 8.0;
    }
    printf("--------------------------------\n");
    for (a=0;a<5;a++)
        printf("E(%4d) = %f (%f usecs)\n", sizes[a], avg[a], avg[a]*USPT);
}

odata_print() {
    printf("--------------------------------\n");
    printf("R test data, then A+U, then F+S, R+S, Tt\n");
    printf("    Bufs Averg min max\n");
    for (a=0;a<5;a++) {
        for (b=0;b<5;b++) {
            printf("%1d>> %4d %6.2f %6.3f %6.3f\n", a, sizes[b],
                sol2[b][a][0], sol2[b][a][1], sol2[b][a][2]);
        }
        printf("\n");
    }
    printf("--------------------------------\n");
    for (a=0;a<5;a++) {
        printf("R(%4d)           = %f (%f usecs)\n",
            sizes[a], sol2[a][0][0], sol2[a][0][0]*USPT);
        printf("(A+U)(%4d)     = %f (%f usecs)\n",
            sizes[a], sol2[a][1][0], sol2[a][1][0]*USPT);
        printf("(F+S)(%4d) = %f (%f usecs)\n",
            sizes[a], sol2[a][2][0], sol2[a][2][0]*USPT);
        printf("(R+S)(%4d)       = %f (%f usecs)\n",
            sizes[a], sol2[a][3][0], sol2[a][3][0]*USPT);
        printf("(2E+2F+2U)(%4d)  = %f (%f usecs)\n",
            sizes[a], sol2[a][4][0], sol2[a][4][0]*USPT);
        printf("\n");
    }
}

sdata_print() {
    printf("\n\n");
    printf("E(n)           = %6.1f + %4.3fn usecs  -", E.base, E.slope);
    printf("Maximum Error: %f usecs\n",E.error);
    printf("R(n)           = %6.1f + %4.3fn usecs  -", R.base, R.slope);
    printf("Maximum Error: %f usecs\n",R.error);
```

```
    printf("(A+U)(n)     = %6.1f + %4.3fn usecs  -", AU.base, AU.slope);
    printf("Maximum Error: %f usecs\n",AU.error);
    printf("(F+S)(n) = %6.1f + %4.3fn usecs  -",FS.base,FS.slope);
    printf("Maximum Error: %f usecs\n",FS.error);
    printf("(R+S)(n)       = %6.1f + %4.3fn usecs  -", RS.base, RS.slope);
    printf("Maximum Error: %f usecs\n",RS.error);
    printf("(2E+2F+2U)(n)  = %6.1f + %4.3fn usecs  -",EFUx2.base,EFUx2.slope);
    printf("Maximum Error: %f usecs\n",EFUx2.error);
    printf("\n");
}

cdata_print() {
    printf("-------------------------\n");
    for (a=0;a<5;a++) printf("E(%4d) = %f\n", sizes[a], avg[a]);
    for (a=0;a<5;a++) printf("R(%4d) = %f\n", sizes[a], sol2[a][0][0]);
}

fit(ret, p, n) struct EQU *ret; double p[]; int n; {
    double aa, bb, cc, dd, ee;
    double err;
    int i;

    aa = bb = cc = dd = 0.;
    for (i=0; i<n; ++i) {
        aa += (double)sizes[i] * (double)sizes[i];
        bb += (double)sizes[i];
        cc += (double)sizes[i]*p[i];
        dd += p[i];
    }
    ee = aa*(double)n - bb * bb;

    ret->base = (aa * dd - bb * cc) / ee;
    ret->slope = (cc*(double)n - bb * dd) / ee;

    ret->error = 0.;
    for (i=0; i<n; ++i) {
        err = ret->base + ret->slope * sizes[i] - p[i];
        if (err < 0.) err = -err;
        if (err > ret->error) ret->error = err;
    }
}
```