Yale University
Department of Computer Science

# A Comparison of CPS, Linda, P4, POSYBL, PVM, and TCGMSG: Two Node Communication Times

Timothy G. Mattson, Craig C. Douglas, and Martin H. Schultz

YALEU/DCS/TR-975
May, 1993

This work was supported in part by the Office of Naval Research (grant N00014-91-J-1576).

# A COMPARISON OF CPS, LINDA, P4, POSYBL, PVM, AND TCGMSG: TWO NODE COMMUNICATION TIMES*

TIMOTHY G. MATTSON , CRAIG C. DOUGLAS AND MARTIN H. SCHULTZ

**Abstract.** In this paper, we compare simple, two node communication times for a number of distributed computing programming environments. For each environment, round trip communication times for a ping/pong program are considered. The times were obtained using two SPARCstation 1 workstations on an isolated ethernet LAN.

**Key words.** Communication, CPS, C–Linda, P4, POSYBL, PVM, TCGMSG

**AMS(MOS) subject classifications.** 65Y05

**1. Introduction.** Distributed computing has emerged as a dominant force in high performance computing. In response, a number of programming environments have emerged that make distributed computing available to the application programmer. With multiple environments, the question naturally arises, which environment is the best?

A complete and general answer to this question does not exist. Which environment is the "best" depends on the specific application as well as on a programmer's skills and personal tastes. Numerous issues including portability, ease of use, and efficiency must be considered to really deal with this question.

Given the impossibility of deriving a general pecking order for distributed computing environments, comparisons usually emphasize a single trait: runtime efficiency. While providing only one part of the picture, this approach is valid as efficiency is one of the few measures that can be applied objectively. The results from these studies, however, have been unsatisfactory and contradictory. Attempts to reproduce the comparisons is usually difficult if not impossible due to insufficient explanations of how the comparisons were carried out. The result is a general state of confusion regarding the runtime efficiency of various programming environments for distributed computing.

In this paper, we attempt to rectify this situation by carrying out "painstakingly correct" comparisons of:

- CPS
- C–Linda
- P4
- POSYBL
- PVM
- TCGMSG

In every case, we present detailed code fragments and describe our procedures so any group can reproduce our results. Furthermore, we carried out the comparisons on an isolated workstation network so effects due to competing processes and network traffic could be controlled.

Each of these environments uses one of three TCP/IP techniques for actually transferring information between computer nodes:

* Department of Computer Science, Yale University, P. O. Box 2158 Yale Station, New Haven, CT 06520-2158. This work was supported in part by the Office of Naval Research, grant N00014-91-J-1576. Yale University Department of Computer Science Research Report YALEU/DCS/TR-975, May, 1993.

- Point-to-point A processor opens a direct connection to another processor, sends data, and then optionally closes the connection (e.g., C–Linda, P4, POSYBL, and PVM).
- Open-a-crossbar All processors can send data to any other processor without starting a new connection (e.g., TCGMSG).
- Ftp A processor starts an ftp session with another processor, sends data, and then terminates the connection (e.g., CPS).

One might be tempted to deduce a programming environment "pecking order" based solely on this study. This would be a serious mistake. The experiments described here only test conflict free communication between two nodes. This is a highly artificial arrangement and extrapolation of these results to the more complicated communication patterns encountered in application programs is unclear. Furthermore, runtime efficiency is *not always* the most important issue for the application programmer. In the course of a program's lifetime, matters pertaining to debugging and maintaining a code can be far more important than runtime efficiency.

The body of this paper begins with a general description of the experimental setup: both the hardware and the common software elements. We then provide a very brief description of each programming environment along with code fragments taken from the test programs. Finally, the results of the timing experiments are presented and briefly analyzed. The paper closes with a number of appendices detailing the raw data and comparisons within a particular programming system.

**2. Experimental Setup.** The experiments described in this study were designed to measure the communication time between two nodes on a local area network. We wanted to eliminate any side effects due to network activity or competing processes running on the workstations. Therefore, we placed two SPARCstation 1 workstations in a room and connected them together using ethernet. Prior to each measurement, the process status was checked on each node to assure that no user processes were executing.

All programs were compiled with the standard SUN OS 4.1.3 C compiler. No optimization switches were set, though we did investigate variation of compiler optimization switches and found that they made no impact on the measured times.

The timings were obtained by measuring the round trip communication time in a "ping/pong" program. A message was sent from one node to the other and then back again. Code fragments are given for each case to show specifically how the communication routines within each package were used. With some of the environments, only one routine was sufficient to implement the ping/pong program (e.g., P4 and TCGMSG). The remainder required two routines. We always refer to the ping code as the *master fragment* and the pong code as the *worker fragment*, independent of the actual number of routines.

Every program used the same procedures to access the system clock. This `double function`, named `wtime()` called the standard UNIX function, `gettimeofday()`:

```
#define  USEC_TO_SEC   1.0e-6
double wtime()
{
   double time_seconds;
   struct timezone tzp;
   struct timeval  time_data;
   tzp.tz_minuteswest = 0;  tzp.tz_dsttime     = 0;
```

```
    gettimeofday(&time_data,&tzp);
    time_seconds  = (double) time_data.tv_sec;
    time_seconds += (double) time_data.tv_usec * USEC_TO_SEC;
    return time_seconds;
}
```

Since we were interested in elapsed time, not CPU time, this is obviously the correct function to use for timings.

The average overhead associated with the calls to the timing function, `wtime()`, was computed to use as a correction factor to the recorded times. This average and the standard deviation were computed with the code fragment:

```
for(sum_t=0.0, sum_t2=0.0, iters = 100; iters-- ; ) {
    t0     = wtime();
    twtime = wtime();
    twtime -= t0;
    sum_t += twtime;               /* accumulate sum of times   */
    sum_t2 += twtime * twtime;     /* accumulate sum of squares */
}
ave_time = sum_t/iterations;
std_dev = (sum_t2-((sum_t*sum_t)/(double)iterations))/(iterations-1);
std_dev = sqrt(std_dev);
```

This overhead was computed within each test program and was always insignificant (on the order of 0.13 milliseconds) compared to the measured round-trip communication times.

Each programming environment was tested by considering 100 iterations for the round trip communication. Each iteration was separately timed and corrected for the overhead associated with calling the clock routine. Once these individual timings were collected, each program called the same statistics analysis routine which found the following values:

- average.
- standard deviation.
- median.
- minimum.
- maximum.

In addition, the iteration that resulted in the minimum or the maximum communication time was reported.

In the following subsections, we provide the key loops from each of the timing programs. This should allow one to unambiguously reproduce each program. For the sake of these subsections, the following data declarations apply:

```
long buff_size_bytes; /* length of message to bounce in bytes  */
long iterations=100;  /* Number of iterations to time */
long iters;           /* Loop index for iterations count */
long *buffer;         /* buffer to bounce from master to worker */
long *incoming;       /* buffer to bounce from worker to master */
long iters;           /* iteration loop index */
double *tp;           /* array of times for each iteration */
```

```
double wtime();       /* Wall Time in Seconds */
double t0, twtime;    /* initial time and timer overhead */
int other;            /* Node ID of "other" node*/
long size;            /* Size of an incomming message */
```

Additional declarations will be provided as needed or if they differ from the above.

**2.1. CPS release 2.7.2 .** CPS [Fausey] (Cooperative Processes Software) is a parallel programming environment designed to support the RISC processor farms at the Fermi National Accelerator Laboratory. CPS provides constructs to support message passing, remote procedure calls, process synchronization, bulk data transfers, and batch processing queues. These last two points make CPS unique among the programming environments considered in this paper.

CPS is optimized for applications requiring asynchronous I/O of large data blocks from devices operating over a large range of communication bandwidths, from high speed disks to relatively slow tape drives. Hence, the tests carried out in this study are well outside of the domain for which the CPS design was optimized.

The CPS program included a common main program and two procedures to carry out the actual timing: `ping()` and `pong()`. The `main()` routine served a number of functions included memory allocation and management of I/O. As with the other systems, we don't present these details. The following code fragment just includes the operations unique to setting up the CPS calculation.

```
/* Some important global declarations */
int  (*poing_funcs[])() = { ping, pong };
int  (*stop_funcs[])()  = { acp_stop_process, acp_stop_job };
int  buffer_no[]        = { 1, 2 };
int  *buffer;

main ( argc, argv )
int     argc;
char    **argv;
{
    int     actual_buffer_size;
    int     buffer_size;
    int     p1, p2, p3, p4, p5, pgm;

                          /* The program is started up so this */
    pgm = atoi( *++argv); /* 0 on one node and 1 on the other  */

    acp_init();              /* Startup CPS */

    p1 = actual_buffer_size;              /* Declare buffers  */
    p2 = buffer_no[pgm];                  /* to be externally */
    acp_declare_block( buffer, &p1, &p2 ); /* accessible      */

    p1 = ACP_ALL_PROCESSES;     /*  Synchronmize the  */
    p2 = 3;                     /*  two processes     */
    acp_sync( &p1, &p2 );
```

```
    /* call ping or pong depending on value of pgm */
    poing_funcs[pgm]( pgm, iterations, actual_buffer_size );

    /* Shut down the process or job as a whole (based on pgm) */
    p1 = 1;
    acp_sleep( &p1 );
    stop_funcs[pgm]();

}
```

The master fragmant is a ping procedure provided the origination point in the round-trip communication and therefore included all of the timing calls.

```
    for( iters = iterations ; iters-- ; ) {
        t0 = wtime();
        p1 = buffer_no[1-pgm];
        p3 = actual_buffer_size;
        p4 = buffer_no[1-pgm];
        p5 = 0;
        acp_send( &p1, buffer, &p3, &p4, &p5 );
        p1 = 1;
        p2 = buffer_no[pgm];
        acp_wait_for_data(&p1,&p2);
        *tp = wtime();
        *tp++ -= ( t0 + twtime);
    }
```

Since the buffers were declared to be externally accessible, they are visible at the user program level and a separate buffer packing/unpacking is not explicitly required. The worker fragment is a pong procedure:

```
    for( iters = iterations ; iters-- ; ) {
        p1 = 1;
        p2 = buffer_no[pgm];
        acp_wait_for_data(&p1,&p2);
        p1 = buffer_no[1-pgm];
        p3 = actual_buffer_size;
        p4 = buffer_no[1-pgm];
        p5 = 0;
        acp_send( &p1, buffer, &p3, &p4, &p5 );
    }
```

**2.2. C–Linda release 2.5.** Linda [Carriero] is an associative, virtual shared memory system. C–Linda's operations act upon this memory to provide the process management, synchronization, and communication functionality required to control MIMD computers. The version of Linda we used is produced and commercially supported by Scientific Computing Associates, Incorporated.

The test program consisted of a master and a worker procedure. The master code contained the timing loop:

```
eval(" Timing worker",worker(buffer_size,iterations));

in(" Worker alive?", ?flag);      /* synchronize processes */

for( iters = iterations ; iters-- ; ) {
   t0 = wtime();

   out("ping", buffer:buffer_size);
   in("pong", ? buffer: );

   *tp = wtime();
   *tp++ -= ( t0 + twtime);
}
```

The corresponding code in the procedure, `worker()` is:

```
out (" Worker alive?", 1);      /* synchronize processes */

for( iters = iterations ; iters-- ; ) {
   in ("ping", ?buffer:);
   out("pong", buffer:buffer_size);
}
```

Network-Linda programs are initiated by a program called **ntsnet**. The **ntsnet** utility [SCA] is very flexible and supports a number of command line options that can effect the programs behavior. We varied the relevant options and settled on the following command line:

```
ntsnet -h -p /tmp -d time_linda
```

where **time_linda** is the timing program executable. Options to vary tuple rehashing had no significant impact on the program's execution.

**2.3. P4 release 1.2.** P4 [Butler] is a distributed computing environment providing constructs to program a number of multiprocessor systems. P4 uses monitors for shared memory systems, message passing for distributed memory systems, and includes support for computing across clusters of shared memory computers. It was produced at Argonne National Laboratory as a follow on to the m4 project [Boyle].

For these code fragments, we need to define some additional message identification variables:

```
int MTYP_4 = 4;        /* message Type for P4 */
int MTYP_5 = 5;        /* message Type for P4 */
```

The P4 program was coded in an SPMD (single program multiple data) structure with two parts: a master and a slave. The top level structure of the program including the master portion of the code follows:

```
p4_initenv(&argc,argv);        /* both nodes call this */
```

```
if (p4_get_my_id() == 0)
/**************************
* Node 0  (Master)
**************************/
{
   p4_create_procgroup();

   buffer   = (long *) p4_msg_alloc (buff_size_bytes);
   incoming = (long *) p4_msg_alloc (buff_size_bytes);

   other = 1;                  /* Node id of worker */
   p4_global_barrier(MTYP_3); /* synchronize processes */

   for( iters = iterations ; iters-- ; ) {
      t0 = wtime();

      p4_sendb (MTYP_4, other,  buffer, buff_size_bytes);
      p4_recv (&MTYP_5, &other, &incoming, &size);

      *tp = wtime();
      *tp++ -= ( t0 + twtime);
   }
   p4_msg_free (buffer);
   p4_msg_free (incoming);
}
else
{
/**************************
* Node 1  (Worker)
**************************/
   slave();
}
```

The worker procedure **slave()** included the following code:

```
my_id  = p4_get_my_id();
buffer = (long *) p4_msg_alloc (buff_size_bytes);

p4_global_barrier(MTYP_3); /* synchronize processes */
other = 0;                 /* Node id of master*/
for( iters = iterations ; iters-- ; ) {
   p4_recv (&MTYP_4, &other, &buffer, &size);
   p4_sendb (MTYP_5, other,  buffer, buff_size_bytes);
}
p4_msg_free (buffer);
```

**2.4. POSYBL release 1.102.** POSYBL [Schoinas] is a public domain associative virtual shared memory system and is a simplified clone of the C–Linda pro-

gramming environment. It was developed at the University of Crete. POSYBL is implemented strictly in terms of a runtime library and therefore can not utilize the optimizations possible with compiler-based Linda systems.

As with the C–Linda program, the POSYBL program was divided into master and worker procedures. The master program has as its timing kernel:

```
int len;

eval_1("#3/posybl_worker", NULL );      /* create worker */

in(lstring("Worker alive?"),qlint(&is_it_alive)); /* synchronize */

for( iters = iterations ; iters-- ; ) {
    t0 = wtime();

    out( lstring("ping"), lnint(buffer,buffer_size) );
    in( lstring("pong"), qlnint(&buffer,&len) );

    *tp = wtime();
    *tp++ -= ( t0 + twtime);
 }
```

The worker procedure, `posybl_worker()`, has the analogous code:

```
out(lstring("Worker alive?"),lint(TRUE)); /* synchronize */
for( iters = iterations ; iters-- ; ) {
    in ( lstring("ping"), qlnint(&buffer,&len) );
    out( lstring("pong"), lnint(buffer,buffer_size) );
 }
```

Notice that these commands require the user to notify the POSYBL runtime library of the types of each object, but are otherwise identical to the C–Linda programs.

**2.5. PVM release 2.4.1.** PVM [Sunderam] has established itself as the *de facto* standard for message passing programming environments for distributed computing. It has been particularly designed to handle heterogeneous networks. It has been developed principally at Oak Ridge National Laboratory and the University of Tennessee.

PVM 2.4.1 includes two classes of message passing routines. The first, `snd/rcv` passes all messages through intermediate daemons. This had the advantage of better scalability, but at the price of substantial additional overhead. The more efficient method, `vsnd/vrcv`, uses direct TCP socket connections between communicating processes and is considerably faster.

Regardless of the basic message passing routines utilized, PVM differs from all other systems we studied in that the communication buffers must be explicitly packed and unpacked. Therefore, to be consistent with the other environments, we included this buffer packing time in the round trip communication time.

Because of these options, it is possible for various groups to report drastically different results with a PVM comparison. We report the results for the `vsnd/vrcv` with buffer packing in the main body of this study and include the other timings for some of the other PVM options in Appendix B.

A code fragment from the PVM timing program follows. It is divided into two parts, a `master` and a `worker`. First we present the `master` code.

```
#define MSGTYPE 1000

int buff_sz= (int) buff_size_bytes;

id = enroll ("pvm_time");
initiate ("worker", "SUN4");

waituntil ("synchcalled");

for (iters = iterations; iters--; )
{
  t0 = wtime ();
  initsend ();
  stat = putbytes ((char *) buffer,  buff_sz);
  vsnd ("worker", 0, MSGTYPE);
  vrcv (MSGTYPE);
  stat = getbytes((char *) buffer, buff_sz);
  *tp = wtime ();
  *tp++ -= (t0 + twtime);
}
leave ();
```

The `worker` was a separate program and contained the code:

```
id = enroll ("worker")

ready ("synchcalled");   /* synchronize with the master */

for (iters = iterations; iters--; )
{
  vrcv (MSGTYPE);
  stat = getbytes((char *) buffer, buff_sz);
  initsend();
  vstat = putbytes((char *) buffer, buff_sz);
  snd ("pvm_time", 0, MSGTYPE);
}
leave ();
```

In the course of this study, PVM 3.0 and 3.1 were released. PVM 3.x includes substantial extensions to PVM's functionality and an entirely new application program interface. We did not time PVM 3.x, however, since the release available during this study did not include fully optimized message passing routines. The message passing routines in PVM 3.1, however, use the `vsnd/vrcv` communication mechanism and therefore should match our `vsnd/vrcv` results.

**2.6. TCGMSG release 4.02.** TCGMSG [Harrison] (Theoretical Chemistry Group Message passing system) is a simple message passing system that has risen to a po-

sition of prominence among computational chemists. It is very efficient for the two node experiments we conducted with communication taking place over direct, point-to-point TCP/IP sockets. It was developed initially at Argonne National Laboratory and now at Pacific Northwest Laboratory.

The TCGMSG program was structured as an SPMD program with the kernel of the timing program given by:

```
long MTYP_1 = 1;
long MTYP_2 = 2;

PBEGIN_(argc, argv);
id = NODEID_();

SYNCH_(&MTYP_1);    /* synchronize processes */

if (id == 0L) {
/**************************
* Node 0  (Master)
**************************/
  other = 1L;     /* id of other node */
  for( iters = iterations ; iters-- ; ) {
     t0 = wtime();

     SND_ (&MTYP_2, buffer, &buff_size_bytes, &other, &SYNC);
     RCV_ (&MTYP_2, buffer, &buff_size_bytes, &lenmes,
        &other, &nodefrom, &SYNC);

     *tp = wtime();
     *tp++ -= ( t0 + twtime);
  }
}
else {
/**************************
* Node 1  (Worker)
**************************/
   other = 0L;     /* id of other node */
   for( iters = iterations ; iters-- ; ) {
      RCV_ (&MTYP_2, buffer, &buff_size_bytes, &lenmes,
        &other, &nodefrom, &SYNC);
      SND_ (&MTYP_2, buffer, &buff_size_bytes, &other, &SYNC);
   }
}
PEND_();
```

**3. Results.** The key results from this project are given in Table 3.1, which contains the average round-trip communication times (in milliseconds) versus the message size for each system. For more details about the performance of individual systems, refer to the raw data in the Appendix A.

The results show clear and consistent performance differences for messages rang-

TABLE 3.1
*Average round trip times in milliseconds*

| Size | TCGMSG | P4 | PVM | C–Linda | POSYBL | CPS |
|---|---|---|---|---|---|---|
| 100 | 3.6 | 4.9 | 5.7 | 7.9 | 15.9 | 1031.7 |
| 400 | 4.9 | 5.2 | 6.7 | 9.1 | 17.6 | 1031.6 |
| 1,000 | 5.9 | 6.3 | 9.2 | 10.9 | 19.4 | 1032.0 |
| 4,000 | 12.6 | 15.4 | 17.7 | 21.1 | 30.5 | 1042.2 |
| 10,000 | 23.4 | 32.8 | 42.5 | 53.6 | 64.3 | 1060.8 |
| 40,000 | 79.9 | 123.4 | 147.3 | 168.9 | 273.0 | 1083.1 |
| 100,000 | 201.6 | 308.1 | 356.3 | 389.1 | 1261.8 | 1161.9 |
| 400,000 | 794.2 | 1213.4 | 1383.3 | 1491.7 | 8466.3 | 1556.3 |
| 1,000,000 | 1978.0 | 3030.5 | 3479.3 | 3711.5 | — | — |

ing in size from 100 bytes to one megabyte. In other words, even at the largest message sizes the ethernet-induced bandwidth limitations did not dominate the communication time. TCGMSG was significantly faster for all message sizes. P4, PVM and C–Linda (in that order) represent a middle range in performance. Finally, CPS and POSYBL were the slowest systems and even failed for the largest message sizes.

The CPS times include startup costs on the order of half a second leading to relatively flat performance across the full range of messages. CPS was designed to fill much broader communication needs than the other programming environments we studied and was optimized for asynchronous transfer of large data blocks with potentially slow, tape I/O systems. Under these circumstances, the observed startup costs for CPS are insignificant.

A detailed analysis of the performance differences is beyond the scope of this paper. It is clear, however, that the management of message buffers at either end of the communication plays a major role in the overall communication performance. This follows from the fact that systems using identical network protocalls (TCGMSG, P4 and PVM) displayed very different results.

It is important to note that two node, point-to-point communication tests are an overly simple way to compare programming environments. More complicated communication patterns found in actual applications are essential to make a fair and complete comparison. We intend to study such cases in a future paper.

Regardless of the programming environment, communication across LAN networks, even in the case of FDDI, is slow relative to computation. Any application that maps well onto a network distributed computer must be coarse grained. Hence, communication time must play a minor role in the application's overall performance making the differences seen here less significant in terms of an application's overall performance. Therefore, issues not addressed in this paper such as ease of use and debugging support are critical when selecting a programming environment.

REFERENCES

[Boyle] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*, Hold, Rinehart, and Winston, 1987.

[Carriero] N. Carriero and D. Gelernter, *How to Write Parallel Programs: A First Course.* (Cambridge: MIT Press, 1990).

[SCA] Scientific Computing Associates, Inc. "C–Linda Reference Manual", (1992).

[Butler] R. Butler and E. Lusk, "User's Guide to the p4 Programming System", Argonne National Laboratory technical report ANL-92/17, (1992).

[Sunderam] V. S. Sunderam, "PVM: A Framework for Parallel Distributed Computing", Concurrency: Practice and Experience, Vol 2, pages 315-339, (1990).

[Schoinas] G. Schoinas, "Issues on the implementation of Programming SYstem for distributed applications", University of Crete draft technical report, (1992).

[Harrison] R. J. Harrison,"Portable Tools and Applications for Parallel Computers", International Journal of Quantum Chemistry, Vol 40, 847-863, (1991).

[Fausey] M. Fausey, F. Rinaldok, S. Wolbers, D. Potter, B. Yeager, "CPS and CPS Batch Reference Guide", Fermi National Accelerator Laboratory, GA0008, (1992).

**A. Raw Data.** In this appendix, we present the raw data for each system studied. For each table, *Size* refers to the size of each message in bytes. All time units are in milliseconds and refer to the round trip communication time. Finally, the quantity in parenthesis in the *Minimum* and *Maximum* columns refers to the iterations that resulted in the minimum or maximum time.

During the course of this investigation, a question about different major releases of both C–Linda and PVM arose. In Tables A.2–A.3, raw data for two releases of C–Linda (2.4.6 and 2.5) are presented.

PVM 2.4.1 has two methods for message passing and separates communication from the buffer manipulation. Consequently, PVM benchmarks can vary widely. Tables A.6–A.7 include buffer manipulation while Table A.8 excludes it. Note that PVM 3.1 using vsnd/vrcv should behave similarly to Table A.6.

TABLE A.1
*CPS round trip times in milliseconds*

| Size | Average | Std dev | Median | Minimum | | Maximum | |
|---|---|---|---|---|---|---|---|
| 100 | 1031.7 | 11.4 | 1029.9 | 1029.1 | (19) | 1141.9 | (0) |
| 400 | 1031.6 | 11.5 | 1029.9 | 1029.0 | (14) | 1142.0 | (0) |
| 1,000 | 1032.0 | 13.3 | 1029.9 | 1029.0 | (35) | 1142.7 | (0) |
| 4,000 | 1042.2 | 22.3 | 1039.9 | 1029.7 | (92) | 1258.2 | (0) |
| 10,000 | 1060.8 | 100.5 | 1049.9 | 1039.6 | (17) | 2049.9 | (31) |
| 40,000 | 1083.1 | 12.9 | 1079.9 | 1079.2 | (31) | 1193.7 | (0) |
| 100,000 | 1161.9 | 12.2 | 1159.9 | 1158.9 | (26) | 1269.7 | (0) |
| 400,000 | 1556.3 | 74.6 | 1539.9 | 1509.8 | (42) | 2199.8 | (22) |

TABLE A.2
*C-Linda 2.50 round trip times in milliseconds*

| Size | Average | Std dev | Median | Minimum | | Maximum | |
|---|---|---|---|---|---|---|---|
| 100 | 7.9 | 0.3 | 7.8 | 7.7 | (50) | 10.2 | (99) |
| 400 | 9.1 | 0.9 | 8.9 | 8.7 | (18) | 16.2 | (1) |
| 1,000 | 10.9 | 0.5 | 10.8 | 10.6 | (89) | 15.1 | (3) |
| 4,000 | 21.1 | 0.6 | 21.0 | 20.7 | (81) | 25.4 | (60) |
| 10,000 | 53.6 | 1.2 | 53.2 | 52.5 | (60) | 61.0 | (0) |
| 40,000 | 168.9 | 11.0 | 164.6 | 162.7 | (59) | 220.6 | (23) |
| 100,000 | 389.1 | 14.9 | 384.3 | 382.2 | (85) | 460.6 | (23) |
| 400,000 | 1491.7 | 48.6 | 1478.3 | 1470.4 | (54) | 1770.5 | (0) |
| 1,000,000 | 3711.5 | 201.5 | 3673.4 | 3651.3 | (22) | 5627.6 | (0) |

TABLE A.3
*C-Linda 2.4.6 round trip times in milliseconds*

| Size | Average | Std dev | Median | Minimum | | Maximum | |
|---|---|---|---|---|---|---|---|
| 100 | 9.4 | 0.2 | 9.3 | 8.1 | (99) | 9.9 | (40) |
| 400 | 10.4 | 0.2 | 10.4 | 9.2 | (99) | 11.0 | (64) |
| 1,000 | 12.7 | 2.7 | 12.0 | 11.8 | (38) | 32.5 | (91) |
| 4,000 | 22.9 | 1.0 | 22.5 | 21.5 | (81) | 27.2 | (86) |
| 10,000 | 52.6 | 1.1 | 52.3 | 51.6 | (20) | 59.9 | (0) |
| 40,000 | 163.9 | 5.8 | 162.2 | 160.8 | (13) | 199.9 | (26) |
| 100,000 | 382.3 | 11.5 | 378.7 | 376.6 | (40) | 449.5 | (0) |
| 400,000 | 1471.8 | 39.5 | 1460.7 | 1451.0 | (32) | 1704.1 | (0) |
| 1,000,000 | 3652.2 | 87.5 | 3625.7 | 3615.3 | (63) | 4434.5 | (0) |

TABLE A.4
*P4 round trip times in milliseconds*

| Size | Average | Std dev | Median | Minimum | | Maximum | |
|---|---|---|---|---|---|---|---|
| 100 | 4.9 | 0.1 | 4.9 | 4.8 | (49) | 5.3 | (78) |
| 400 | 5.2 | 0.3 | 5.1 | 5.1 | (35) | 7.3 | (0) |
| 1,000 | 6.3 | 0.2 | 6.2 | 6.1 | (87) | 7.8 | (64) |
| 4,000 | 15.4 | 2.8 | 14.5 | 13.8 | (61) | 35.3 | (96) |
| 10,000 | 32.8 | 1.0 | 32.6 | 31.5 | (7) | 41.7 | (0) |
| 40,000 | 123.4 | 4.4 | 122.8 | 120.4 | (60) | 161.8 | (0) |
| 100,000 | 308.1 | 18.4 | 303.0 | 295.3 | (9) | 402.8 | (0) |
| 400,000 | 1213.4 | 76.1 | 1194.3 | 1184.3 | (99) | 1866.3 | (0) |
| 1,000,000 | 3030.5 | 164.7 | 2989.2 | 2967.5 | (72) | 4555.1 | (0) |

TABLE A.5
*POSYBL round trip times in milliseconds*

| Size | Average | Std dev | Median | Minimum | | Maximum | |
|---|---|---|---|---|---|---|---|
| 100 | 15.9 | 1.2 | 15.9 | 13.9 | (3) | 19.4 | (77) |
| 400 | 17.6 | 1.4 | 17.4 | 15.3 | (6) | 22.4 | (88) |
| 1,000 | 19.4 | 5.2 | 18.8 | 16.3 | (3) | 68.0 | (99) |
| 4,000 | 30.5 | 6.0 | 29.8 | 22.2 | (16) | 79.1 | (99) |
| 10,000 | 64.3 | 10.7 | 62.1 | 52.6 | (4) | 130.0 | (99) |
| 40,000 | 273.0 | 150.5 | 248.5 | 173.8 | (4) | 1471.4 | (68) |
| 100,000 | 1261.8 | 653.5 | 1048.0 | 465.5 | (70) | 4545.4 | (0) |
| 400,000 | 8466.3 | 3601.0 | 7682.6 | 3291.3 | (20) | 19883.4 | (73) |

TABLE A.6
*PVM 2.4.1 vsnd.vrcv round trip times in milliseconds*

| Size | Average | Std dev | Median | Minimum | | Maximum | |
|---|---|---|---|---|---|---|---|
| 100 | 5.7 | 1.1 | 5.5 | 5.4 | (20) | 14.7 | (0) |
| 400 | 6.7 | 0.9 | 6.6 | 6.5 | (98) | 15.1 | (0) |
| 1,000 | 9.2 | 1.9 | 8.5 | 8.0 | (30) | 22.7 | (95) |
| 4,000 | 17.7 | 1.4 | 17.5 | 17.0 | (44) | 29.5 | (0) |
| 10,000 | 42.5 | 9.5 | 40.2 | 39.1 | (83) | 127.4 | (0) |
| 40,000 | 147.3 | 11.7 | 145.0 | 142.3 | (16) | 243.9 | (0) |
| 100,000 | 356.3 | 19.2 | 351.7 | 347.9 | (96) | 498.7 | (0) |
| 400,000 | 1383.3 | 62.5 | 1370.3 | 1356.7 | (45) | 1914.1 | (0) |
| 1,000,000 | 3479.3 | 300.4 | 3424.1 | 3405.9 | (17) | 5944.2 | (1) |

TABLE A.7
*PVM 2.4.1 snd/rcv round trip times in milliseconds*

| Size | Average | Std dev | Median | Minimum | | Maximum | |
|---|---|---|---|---|---|---|---|
| 100 | 14.1 | 0.2 | 14.1 | 13.2 | (6) | 14.8 | (9) |
| 400 | 16.2 | 1.2 | 16.1 | 14.9 | (21) | 27.4 | (78) |
| 1,000 | 18.5 | 0.2 | 18.5 | 17.1 | (61) | 19.2 | (86) |
| 4,000 | 43.1 | 1.1 | 42.9 | 41.9 | (52) | 51.4 | (0) |
| 10,000 | 71.6 | 2.3 | 71.9 | 67.1 | (50) | 87.7 | (0) |
| 40,000 | 246.4 | 12.5 | 242.1 | 229.8 | (26) | 298.2 | (53) |
| 100,000 | 585.0 | 31.3 | 577.1 | 542.7 | (11) | 732.2 | (75) |
| 400,000 | 2235.5 | 66.8 | 2231.2 | 2125.1 | (17) | 2716.9 | (0) |
| 1,000,000 | 5652.2 | 591.7 | 5586.1 | 5400.9 | (80) | 11146.0 | (1) |

TABLE A.8
*PVM 2.4.1 snd/rcv, no buffers in milliseconds*

| Size | Average | Std dev | Median | Minimum | | Maximum | |
|---|---|---|---|---|---|---|---|
| 100 | 13.8 | 4.0 | 13.3 | 11.9 | (85) | 53.3 | (0) |
| 400 | 12.0 | 0.3 | 12.0 | 10.5 | (96) | 13.6 | (11) |
| 1,000 | 15.4 | 0.2 | 15.3 | 14.5 | (4) | 15.9 | (88) |
| 4,000 | 27.6 | 3.1 | 26.6 | 25.4 | (36) | 46.2 | (28) |
| 10,000 | 38.3 | 0.9 | 38.1 | 37.2 | (35) | 45.3 | (0) |
| 40,000 | 112.8 | 2.4 | 113.4 | 108.9 | (31) | 127.4 | (0) |
| 100,000 | 257.5 | 18.0 | 252.9 | 249.2 | (83) | 365.4 | (0) |
| 400,000 | 980.2 | 29.2 | 970.0 | 962.7 | (49) | 1105.5 | (1) |
| 1,000,000 | 2406.2 | 70.6 | 2385.5 | 2366.9 | (21) | 2970.3 | (0) |

TABLE A.9
*TCGMSG round trip times in milliseconds*

| Size | Average | Std dev | Median | Minimum | | Maximum | |
|---|---|---|---|---|---|---|---|
| 100 | 3.6 | 0.5 | 3.4 | 3.3 | (63) | 6.8 | (64) |
| 400 | 4.9 | 0.3 | 4.7 | 4.6 | (7) | 6.7 | (1) |
| 1,000 | 5.9 | 0.3 | 5.7 | 5.6 | (18) | 6.7 | (2) |
| 4,000 | 12.6 | 0.4 | 12.4 | 12.1 | (87) | 14.2 | (7) |
| 10,000 | 23.4 | 0.8 | 23.2 | 22.8 | (25) | 29.8 | (81) |
| 40,000 | 79.9 | 0.6 | 79.8 | 78.8 | (60) | 83.9 | (15) |
| 100,000 | 201.6 | 6.1 | 199.8 | 197.6 | (5) | 239.7 | (9) |
| 400,000 | 794.2 | 20.9 | 789.8 | 778.7 | (48) | 917.0 | (69) |
| 1,000,000 | 1978.0 | 40.1 | 1964.0 | 1951.2 | (26) | 2101.4 | (91) |