PARALLEL PROGRAMMING SYSTEMS FOR WORKSTATION CLUSTERS*

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

Abstract. In this paper, we describe experiments comparing the communication times for a number of different network programming environments on isolated 2 and 4 node workstation networks. In addition to simplified benchmarks, a real application is used in one of these experiments.

From our results, it is clear that the cost of buffer management at either end of the communication is more important than originally expected. Furthermore, as communication patterns become more complex, the performance differences between these environments decreased substantially. When we compared timings for an actual application program, the differences essentially disappeared. This shows the danger of relying solely on simplified benchmarks.

Key words. Parallel computing, Communication, C–Linda, P4, POSYBL, PVM, TCGMSG

AMS(MOS) subject classifications. 65Y05

1. Introduction. A number of programming environments exist that make distributed computing available to the application programmer. While it is natural to try, selecting a single best environment is impossible. The evaluation depends on issues related to portability, ease of use, and efficiency as well as personal matters such as programmer skill and individual taste.

Given the impossibility of a general ranking of distributed computing environments, comparisons usually emphasize a single trait, namely run time efficiency. Focusing only on the efficiency is much too narrow for a complete evaluation. However, it is a useful factor that can be applied objectively and performance is one of the major reasons for considering parallel computing in the first place. Past comparisons of programming environment efficiencies have often been contradictory. Attempts to reproduce these experiments are usually difficult if not impossible due to insufficient descriptions of how the comparisons were completed. The result is a general state of confusion regarding the run time efficiency of various programming environments for distributed computing.

In this paper, we attempt to rectify this situation by carrying out painstakingly careful benchmarks. In every case, detailed code fragments are provided in the appendices. Complete codes are available electronically on the Internet in the benchmark directory on any *netlib* code repository. Our procedures are described in sufficient detail so that any group can reproduce our results. Furthermore, we did the comparisons on

* This work was supported in part by International Business Machines and the Office of Naval Research, grant N00014-91-J-1576. Yale University Department of Computer Science Research Report YALEU/DCS/TR-975, August, 1993.

† Department of Computer Science, Yale University, P. O. Box 2158 Yale Station, New Haven, CT 06520-2158. E-mail: *douglas-craig@cs.yale.edu*.

‡ Intel Corporation, SuperComputer Systems Division, C06-09 Building, Zone 8, 14924 N.W. Greenbrier Parkway, Beaverton, OR 97006. E-mail: *tgm@ssd.intel.com*.

§ Department of Computer Science, Yale University, P. O. Box 2158 Yale Station, New Haven, CT 06520-2158. E-mail: *schultz-martin@cs.yale.edu*.

Table 1
Systems tested

System	Citation	How to get
C–Linda	[3], [9]	Send e-mail to *software@sca.com* for more details.
P4	[2]	Available by anonymous ftp from *info.mcs.anl.gov* in the directory *pub/p4*.
POSYBL	[8]	Available by anonymous ftp from *ariadne.csi.forth.gr* in the directory *posybl*.
PVM	[11]	Available by anonymous ftp from *netlib2.ornl.gov* in the directory *pvm3*.
TCGMSG	[5]	Available by anonymous ftp from *ftp.tcg.anl.gov* in the directory *pub/tcgmsg*.

isolated workstation networks so effects due to competing processes and network traffic were controlled and minimized.

The experiments described here cover three different communication scenarios. The two node tests measure conflict free communication between nodes (a simple *ping/pong* program). These low level tests provide information about raw performance, but are highly artificial and extrapolation of these results to actual applications may lead to misleading predictions. Consequently, we considered a more complicated communication pattern looking at simultaneous shifts of data about a ring of four nodes. Taken together, these tests provide a view of the performance of each programming environment studied in this paper. Finally, we measured the performance of the most efficient message passing paradigm (TCGMSG) and virtual shared memory paradigm (C–Linda) for a commercial molecular dynamics application. It is important to note, however, that even though we emphasize communication time, run time efficiency is not always the most important issue for the application programmer. In the course of a program's existence, issues such as ease of use, debugging, and software maintenance can be far more important than run time efficiency.

The programs in §§3–4 were compiled with the standard SUN OS 4.1.3 C compiler. These programs were run on identical SUN SPARCstation 1 workstations connected using an isolated ethernet. The computers had identical, complete file systems and were not running a network file system. Each machine had 8 Mb of random access memory.

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. In addition, we used every programming environment as provided. We made no attempt tune the systems for the actual networks used here. We also used default compiler flag settings when building the various public domain systems.

In §2, a discussion of the different protocols used by the programming environments is presented. In §3, a two node study is discussed. In §4, a four node study is discussed. In §5, a commercial parallel application is discussed. Finally, in §6, we draw three conclusions.

Appendices A and B contain code fragments for the two and four node studies. The code fragments may be of interest to people who wish to reproduce these results or who are curious as to the different programming styles required to do the same thing in the various environments. The raw data that contributes to Tables 2 and 3 is in Appendix C.

2. Network protocols. All of the programming environments studied in this paper have the same goal: the nodes of a workstation cluster are made to act like members of a loosely coupled, parallel computer. In every case, this is done by mapping some higher level model onto low level network protocols. We consider two high level protocols, namely,

1. message passing, and
2. shared virtual memory.

See [10] for definitions of terms like TCP and UDP.

With TCGMSG, point to point TCP sockets are established between every pair of nodes. This is done when the program is initiated and these sockets are not reclaimed in the course of the calculation. We call this approach the *static TCP socket* system (sometimes this is referred to as a TCP socket crossbar). The static TCP sockets method can run into trouble scaling up to large numbers of nodes since the number of open file descriptors per node grows as the twice the number of nodes. (On UNIX systems this is usually not a problem since the kernel can be reconfigured for as many file descriptors as necessary.) However, it is fast, simple, and has the advantage of hiding the start up time of establishing connections between all of the nodes.

PVM and P4 both use *dynamic TCP sockets.* This means they establish a socket between two communicating nodes at run time when they first communicate with each other. The sockets are generally not reclaimed in the course of a computation. This method has the advantage that it will scale better on a large set of nodes as long as none of the processors runs out of file descriptors (as in the static TCP socket case). One disadvantage of dynamic TCP relative to static TCP is that the first communication is significantly slower than subsequent communications (and hard to hide).

PVM and P4 differ in what options they provide the user. With P4 message passing, dynamic TCP sockets is the only option. PVM, however, lets the user choose between TCP sockets and daemon mediated, UDP communication. In PVM 2.4.2, the choice is made by the user when they select snd/rcv (UDP) or vsnd/vrcv (TCP). PVM 3.X only has one set of snd/rcv routines. By default, the system provides the daemon mediated, UDP communication. To get the more efficient TCP socket based communication, the user uses the following call prior to the communication:

```
pvm_advise (PvmRouteDirect);
```

The use of TCP sockets is clearly superior when communication between pairs of nodes takes place many times so that startup costs can be amortized. When two nodes will only communicate a few times, it may be faster to use the daemon based, UDP method. Characterizing these tradeoffs and providing guidelines concerning the conditions under which the two methods should be used has yet to be determined and is an active research area within the PVM group.

The version of C–Linda used employs UDP to communicate directly with the processes involved in the parallel computation. All the processes involved in a computation agree on a UDP port and use it during the computation. C–Linda does not use TCP sockets or an intermediate daemon. Instead, an extra process on each node performs all of the local shared virtual memory management and communicates with the other nodes. This process gains control using a signal handler that initiates a context switch to the tuple space handler.

3. Two node studies. The experiments described in this section 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 connected two SUN SPARCstation 1 workstations using an isolated ethernet. Prior to each measurement, the process status was checked on each node to assure that no user processes were executing.

The timings were obtained by measuring the round trip communication time in a so-called *ping/pong* program: a message was sent from one node to the other and then back again.

Since we were interested in elapsed time, not CPU time, every program accessed the system clock by calling the standard UNIX function *gettimeofday()* for timings.

The average overhead associated with the calls to the timing function was computed to use as a correction factor to the recorded times. 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. The key results from this study are given in Tables 2 and 3. The former contains the average round-trip communication times (in milliseconds) versus the message size for each system. The latter contains the data transfer rates (in megabytes per second).

Table 2 shows clear and consistent performance differences for messages ranging in size from 100 bytes to one megabyte. TCGMSG was significantly faster for all message sizes. P4, PVM and C–Linda (in that order) represent a middle range in performance. Finally, POSYBL was the slowest system and even failed for the largest message size.

A detailed analysis of these performance differences is beyond the scope of this

Table 2
Average round trip times for the two node studies

Bytes in message	Message passing				Virtual shared memory	
	TCGMSG	P4	PVM[1]	PVM[2]	C–Linda	POSYBL
100	3.6	4.9	5.7	14.1	7.9	15.9
400	4.9	5.2	6.7	16.2	9.1	17.6
1,000	5.9	6.3	9.2	18.5	10.9	19.4
4,000	12.6	15.4	17.7	43.1	21.1	30.5
10,000	23.4	32.8	42.5	71.6	53.6	64.3
40,000	79.9	123.4	147.3	246.4	168.9	273.0
100,000	201.6	308.1	356.3	585.0	389.1	1261.8
400,000	794.2	1213.4	1383.3	2235.5	1491.7	8466.3
1,000,000	1978.0	3030.5	3479.3	5652.2	3711.5	—

All times are in milliseconds.
[1] Using vsnd/vrcv.
[2] Using snd/rcv.

Table 3
Average data transfer rates for the two node studies

Bytes in message	Message passing				Virtual shared memory	
	TCGMSG	P4	PVM[1]	PVM[2]	C–Linda	POSYBL
100	.0556	.0408	.0350	.0142	.0254	.0126
400	.1632	.1538	.1194	.0494	.0880	.0454
1,000	.3390	.3174	.2174	.1082	.1834	.1030
4,000	.6350	.5194	.4520	.1856	.3792	.2622
10,000	.8548	.6098	.4706	.2794	.3732	.3110
40,000	1.0012	.6482	.5432	.3246	.4736	.2930
100,000	.9920	.6492	.5614	.3418	.5140	.1586
400,000	1.0074	.6594	.5784	.3578	.5364	.0944
1,000,000	1.0112	.6600	.5748	.3538	.5388	—

All rates are in megabytes per second.
[1] Using vsnd/vrcv.
[2] Using snd/rcv.

paper. It is clear 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 protocols (TCGMSG, P4, and PVM) displayed very different results.

The maximum possible bandwidth we could have measured was 1.25 megabytes per second. It follows from Table 3 that even at the largest message sizes the ethernet-induced bandwidth limitations did not dominate the communication time.

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.

4. Four node studies. Evidence based on applications level studies indicated to us that the network programming systems did not vary significantly in performance. A reasonable inference is that the addition of communication contention would equalize the performance of the various systems. To test this idea, we added two more SPARC-station 1 nodes to the isolated network so that a total of four identical nodes were on the isolated ethernet. We called our four node benchmark the *ring_test* program because it does the following:

- Starts a program on each of the four nodes.
- Constructs an array on each node.
- Each node then passes its array to its neighbor, i.e., the nodes shift the data around the ring.
- Repeats this for some number of shifts.

We varied the size of the messages and considered 100 shifts. We timed these shifts and reported the time and net bandwidth for the process.

We used the same timing routine as in the two node tests. Since multiple shifts were considered, the timing corrections were insignificant and were omitted.

The key results from this study are given in Tables 4 and 5, where we give the best results for the ring test transfer times (in milliseconds) and the average bandwidth (in megabytes per second) as a function of message size for each of the network programming environments. There were a number of different ways of programming this example using either P4 or PVM, which are summarized in Table 6. For reasons that we do not understand, PVM's performance dropped once the message sizes became large, which is summarized using a finer data size sampling in Table 7.

From Table 5 we see that TCGMSG is still the fastest of the systems tested. The ring tests do not maintain the performance differences observed in the two node tests. While the two node tests showed TCGMSG as being approximately 50% faster for the largest message sizes, the four node tests have TCGMSG only 20% faster. We suspect this effect follows from the fact that TCGMSG communication on a network is synchronous. The rest of the network programming environments support some low level overlap of the sends and receives, but TCGMSG must synchronize at each send/receive pair. This effect was not present in the two node tests since that test was inherently synchronous.

Table 4
Times for the four node studies

Bytes in message	Message passing P4[1]	PVM[2]	TCGMSG[2]	Virtual shared memory C–Linda[1]	POSYBL[1]
800	23.8	1.5	1.6	2.1	8.0
8,000	32.0	10.7	7.1	11.5	24.8
80,000	93.9	102.0	75.6	86.9	218.0
800,000	958.0	3139.7	761.5	911.3	—

All times are in milliseconds.
[1] NO-SPLIT (i.e. all sends followed by all receives).
[2] SPLIT (PVM used vsnd/vrcv).

Table 5
Data transfer rates for the four node studies

Bytes in message	Message passing P4[1]	PVM[2]	TCGMSG[2]	Virtual shared memory C–Linda[1]	POSYBL[1]
800	0.0336	0.5212	0.5040	0.3837	0.0996
8,000	0.2498	0.7472	1.1232	0.6952	0.3226
80,000	0.8519	0.7841	1.0583	0.9210	0.3669
800,000	0.8351	0.2548	1.0506	0.8779	—

All rates are in megabytes per second.
[1] NO-SPLIT (i.e. all sends followed by all receives).
[2] SPLIT (PVM used vsnd/vrcv).

TCGMSG required that the communication be split into two parts: half the nodes did a send/receive while the other half did a receive/send. This was required since TCGMSG communication on a network is synchronous. This is referred to in the tables as *SPLIT*. The other option available with the other systems was to post all the sends and then all the receives. This is referred to in the tables as *NO-SPLIT*. Both of these options were considered for P4 and PVM. C–Linda and POSYBL hide low level message passing details from the user so we decided to only investigate the NO-SPLIT algorithm for these environments.

5. Application studies. Both the four node and two node studies use highly artificial benchmarks to compare performance. This approach is valuable in that we can be sure that the comparisons are based on consistent communication patterns. However, these low level tests can be misleading by suggesting performance advantages for one system over another that are not relevant in actual applications.

A suggestion of this effect is present in comparing the two node and four node results. The four node test is more realistic than the two node test in that it more closely resembles the communication context of an actual application.

To study this effect further, a molecular dynamics program (Wesdyn [7]) which

Table 6

P4 and PVM data transfer rates for the four node studies

Bytes in message	P4		PVM vsnd/vrecv		PVM snd/rcv	
	NO-SPLIT	SPLIT	NO-SPLIT	SPLIT	NO-SPLIT	SPLIT
800	0.0336	0.0170	0.5858	0.5212	0.3095	0.2669
8,000	0.2498	0.1453	0.8094	0.7472	0.4405	0.4092
80,000	0.8519	0.5739	0.7967	0.7841	0.4324	0.4324
800,000	0.8351	0.8188	0.1819	0.2548	0.0510	0.0690

All rates are in megabytes per second.

Table 7

More extensive PVM data transfer rates for the four node studies

Bytes in message	PVM[1]
800	0.3095
8,000	0.4405
80,000	0.4324
160,000	0.5389
240,000	0.7960
320,000	0.5349
400,000	0.4418
480,000	0.1872
560,000	0.7954
640,000	0.0791
720,000	0.0683
800,000	0.0510

All rates are in megabytes per second.

[1] NO-SPLIT, snd/rcv.

Table 8
Wesdyn elapsed times

Nodes	TCGMSG	C–Linda
1	1349	1348
2	777	780
3	690	625
4	637	589

All times are in seconds.

uses C–Linda already was recoded [6] using TCGMSG. We ran one of the standard Wesdyn benchmarks (50 minimization steps) on a cluster of IBM RS/6000 model 560 workstations connected by ethernet. The same compiler optimization levels and the same algorithms were used for the two versions of the program.

This application is excellent for this type of comparison. First, it is a real-world application in commercial production usage in the pharmaceutical industry and therefore not an artificial benchmark. Second, the problem is seriously communication bound. The performance on a workstation network saturates the communication bandwidth by the third or fourth node depending on the problem size. Hence, we expect that if differences in communication performance were significant, this application (rather than an embarrassingly parallel, compute bound problem) would expose those differences.

The results of this comparison are given in Table 8. Notice that the TCGMSG program runs at about the same speed as the C–Linda program for small numbers of nodes but by three nodes, it is on the order of 10% slower. Once again, this appears to be due to the fact that TCGMSG communication for workstation clusters is synchronous.

6. Conclusions. In this paper, we have presented a number of benchmarks to compare the performance of a number of important programming environments. Several conclusions can be drawn from this paper.

First, the cost of buffer management at either end of the communication is more important than we originally expected. Rather than all systems performing the same for simple communication patterns and large messages, significant and substantial differences in performance were found.

Second, as communication patterns become more complex, the differences in these environments decreased substantially. With the two node tests, differences between the systems ranged up to 50%. However, for the four node tests, the differences dropped to 5%.

Finally, when we compared timings for an actual application program, the differences essentially disappeared. This shows the danger of relying solely on simplified benchmarks.

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.

Acknowledgments. We would like to thank Adam Begulin, Matt Fausey, Al Geist, Robert Harrison, and Rusty Lusk for helpful comments made during this study.

REFERENCES

[1] 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.

[2] R. Butler and E. Lusk, *User's guide to the p4 programming system*, Tech. Rep. Report ANL-92/17, Argonne National Laboratory, 1992.

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

[4] M. Fausey, F. Rinaldok, S. Wolbers, and B. Y. D. Potter, *Cps and cps batch reference guide*, Tech. Rep. Report GA0008, Fermi National Accelerator Laboratory, Batavia, IL, 1992.

[5] R. J. Harrison, *Portable tools and applications for parallel computers*, Int. J. Quantum Chem., 40 (1991), pp. 847–863.

[6] T. G. Mattson and G. Ravishankar, *Parallel molecular dynamics with wesdyn*. In preparation, 1993, 1993.

[7] G. Ravishankar and S. Swaminathan, *Wesdyn*, Wesleyan University, Middletown, CT, 199x.

[8] G. Schoinas, *Issues on the implementation of programming system for distributed applications*, tech. rep., University of Crete, 1992.

[9] Scientific Computing Associates, Inc., *C–Linda Reference Manual*, New Haven, CT, 1992.

[10] W. R. Stevens, *UNIX Network Programming*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[11] V. S. Sunderam, *Pvm: a framework for parallel distributed computing*, Concurrency: Practice and Experience, 2 (1990), pp. 315–339.

A. Code fragments for the two node studies. The timings for the two node studies in §3 were obtained by measuring the round trip communication time in a so-called *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` (`wtime()`) called the standard UNIX function `gettimeofday()` since we were interested in elapsed time, not CPU time.

```
#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;
 }
```

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 incoming message */
```

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

A.1. C–Linda release 2.5. Linda [3] is an associative, virtual shared memory system. 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 [9] 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.

A.2. P4 release 1.2c. P4 [2] 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 [1].

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);
```

A.3. POSYBL release 1.102. POSYBL [8] is a public domain associative virtual shared memory system and is a simplified clone of the C–Linda programming 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_l("#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.

A.4. PVM release 2.4.1. PVM [11] was specifically designed to handle heterogeneous networks. It has been developed principally at Oak Ridge National Laboratory and the University of Tennessee.

PVM 2.4 includes two classes of message passing routines. The first, `snd/rcv` passes all messages through intermediate daemons. This had the advantage of excellent 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 an appendix.

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.X, however, include the `vsnd/vrcv` communication mechanism and therefore should match our `vsnd/vrcv` results.

A.5. TCGMSG release 4.02. TCGMSG [5] (Theoretical Chemistry Group Message passing system) is a simple message passing system that has risen to a position 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_();
```

B. Code fragments for the four node studies. In the four node tests, we timed the movement of data around a ring of nodes. Each benchmark was coded in an SPMD style and included:

- A program to initiate processes on the nodes of the ring and to input and distribute benchmark data.
- A function (named `worker()`) to call, time and analyze the ring benchmark on each node.
- An SPMD style function (named `ring()`) to do the actual ring communication.

Both Linda systems hide low level communication details so we only considered the most straight forward ring algorithm in which all nodes execute an `out` followed by all nodes executing an `in`. The message passing systems, however, support two different ring communication patterns. In the first pattern, half the nodes `send()` then `receive()` while the other half of the nodes `receive()` then `send()`. We call this the SPLIT method. The other approach is analogous to the one used for Linda; i.e. all nodes `send()` and then all nodes `receive()`. We call this method NO_SPLIT. The network implementation of TCGMSG only supports synchronous communication so we could only use the SPLIT method. For PVM and P4, we considered both.

Regardless of the ring communication pattern, each node individually computed an elapsed time using the timing routine from the two node tests (`wtime()`). The nodes then collectively compute the maximum and minimum times required for the ring communication. The minimum bandwidth is computed and output along with the minimum and maximum times for the benchmark. The following code fragment shows how the bandwidth was computed.

```
#define MB_CONV  1.0e-6              /* convert bytes to Megabytes */
out_results (max_time, num_nodes, size, num_shifts)
double max_time;    /* maximum time for ring test */
int num_nodes;      /* number of nodes in the ring */
int size;           /* size of the message passed around the ring */
int num_shifts;     /* number of times the message is shifted */
{
    double bandwidth;   /* bandwidth */

    bandwidth = (double)(num_shifts * num_nodes * size *
                         sizeof(double));
    bandwidth = MB_CONV * bandwidth / max_time;
    printf("\n Bandwidth = %5.4f megabytes/second \n",bandwidth);
}
```

In the remainder of this appendix, code fragments will be provided for each of the systems studied with the ring test. Within these code fragments, the following data declarations apply:

```
#define IS_ODD(x) ((x)%2) /* macro to test for an odd int */

int buff_size_bytes; /* Length of vector to sum in bytes  */
double *x;           /* Vector to communicate around the ring*/
double *incoming;    /* Vector to communicate around the ring*/
double t0,wtime();   /* Wall Time in Seconds */
double ring_time;    /* Time for ring comm.  on this node */
double min_time;     /* Time for ring com - min time for all nodes*/
double max_time;     /* Time for ring com - max time for all nodes*/
int id;              /* Node id (id ranges from 0 to num_nodes-1) */
int num_nodes;       /* Number of nodes involved in the test */
int size;          /* Size of double vector used in the summation */
int num_shifts;    /* Size of double vector used in the summation */
```

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

Before considering the code fragments, we have some general observations to make about the coding of these benchmark programs with the various programming environments. First, PVM, POSYBL and C–Linda did not provide the global communication operations required to compute the minimum and maximum times. These functions are time consuming to develop and actually very difficult to do correctly. We don't provide those code fragments in this paper, but direct the interested reader to the benchmark source code available over Internet.

For every system other than C–Linda and POSYBL, separate buffers for the outgoing and incoming message were used. This was done strictly to aid testing. C–Linda provides an easy to use testing and development tool called Tuplescope [9] and therefore, we did not need to use a separate `incoming` buffer. Tuplescope made the C–Linda program far easier to debug than any of the other systems. This benefit carried over to the POSYBL code since the POSYBL program was a straight forward translation of the C-Linda program.

Finally, the PVM program was greatly complicated by the fact that nodes could not be identified by a single integer label. It was further complicated by the fact that instance numbers received from the daemon depended on the state of the daemon. Therefore, the version of the ring test we used lacked in robustness and for a production level application, our solution would be inadequate. A more rigorous solution would require the use of a globally updated array of instance numbers – an approach forced on the programmer in PVM 3.X. We opted not to use this approach since part of our goal was to make each benchmark program as similar as possible. We understand the reasons for this additional complexity (support of heterogeneity), but it is only fair to point out that it substantially complicated the coding and running of the PVM benchmarks.

B.1. C–Linda. The C–Linda program consisted of two modules. The first just handled command line input, `eval'ed` the `worker()` functions, and then becomes a `worker()`:

```
num_nodes   = atoi(*++argv);
size        = atoi(*++argv);
num_shifts  = atoi(*++argv);

for(i=1;i<num_nodes;i++)
     eval("worker",worker(i,num_nodes,size,num_shifts));

worker(0,num_nodes,size,num_shifts);
```

Notice that C–Linda lets one pass scalar arguments to the worker routines through the function argument list. This saves the programmer from having to move that data through explicit interprocess communication mechanism. The second module is the SPMD `worker()` routine:

```
double *work;   /* work space for global operations */

buff_size_bytes = size * sizeof(double);
x = (double *)malloc(buff_size_bytes);
x = (double *)malloc(sizeof(double));
for(i=0;i<size;i++){
   x[i]   = i;
}
```

```
sync(id,num_nodes);   /* synchronize all processes */

t0 = wtime();
ring(x,size,num_nodes,num_shifts,id);
ring_time = wtime() - t0;

min_time = ring_time;
max_time = ring_time;
gdhi(&max_time,1,work,num_nodes,id);
gdlow(&min_time,1,work,num_nodes,id);
if(id == 0)
    out_results(min_time, max_time, num_nodes, size, num_shifts);
```

C–Linda does not provide global communication routines. Therefore, we had to write the barrier routine, `sync()` and the global combine routines: `gdhi()` to find the maximum and `gdlow()` to find the minimum. The global combine routines were developed during an earlier project and use a sophisticated binary tree algorithm. They are rather complex and their inclusion is well beyond the scope of this paper. We safely omit them from this discussion as their function is standard and well understood. The next code fragment was taken from the `sync()` function which provided a barrier to synchronize the processes prior to the ring test timing.

```
int sync(id,n)
int n;        /* number of nodes to synchronize     */
int id;       /* node id -- node 0 must be involved */
{
   int count,i;

   if(id == 0) {
      i = inp ("sync ",?count); /* Clean up counter from past   */
      out("sync ",1);           /* runs and initialize this run.*/
   }
   else {
      in("sync ",?count);       /* Grab shared counter.      */
      out("sync ",count+1);     /* Increment and out counter.*/
   }
   rd("sync ",n);        /* Wait for everyone else. */
}
```

The ring communication test within the `ring()` function was quite simple with C–Linda:

```
/* shift messages around the ring of nodes */
```

```
 next = (id+1)%num_nodes;
 for(i=0;i<num_shifts; i++){
    out(" shift message ",next,x:size);
    in (" shift message ",id,?x:len);
 }
```

This is analogous to the NO_SPLIT method used for the message passing systems. While it was far easier to write and debug the basic ring test with C–Linda, the need to develop the three global operations substantially increased the difficulty of writing this program.

B.2. P4. The P4 program used an SPMD style of coding. On node 0, the program started the processes on the other nodes (using the `procgroup` file) loaded input data, and broadcast this data to each of the other nodes. Nodes other than node 0, initialized their environment and received the benchmark data. In all cases, the routines then called the same worker module.

The code fragments in this section required some additional parameters. Primarily, these were required to deal with the message types and the extra communication of the benchmark parameters:

```
int MsgType1   = 1;
int MsgType2   = 2;
int *msg, msg_len;
int zero = 0;
int buff_size_bytes;
FILE *fp;
```

The code fragment for the programs main routine follows:

```
p4_initenv(&argc,argv);
p4_create_procgroup();
id = p4_get_my_id();

if(id == 0){
   fp =  fopen("ring.dat","r");    /* P4 uses command line args   */
   fscanf(fp,"%d",&size);          /* so we found it easier to    */
   fscanf(fp,"%d",&num_shifts);    /* load parameters from a file */
   fclose(fp);                     /* rather than the command line*/
   msg_len = sizeof(int);
   p4_broadcast(MsgType1, (char *) &size,       msg_len);
   p4_broadcast(MsgType2, (char *) &num_shifts, msg_len);
}
else{
```

```
    msg = (int *) NULL;    /* tell P4 to allocate a comm. buffer */
    p4_recv(&MsgType1, &zero, (char **) &msg, &msg_len);
    size = *msg;

    msg = (int *) NULL;    /* tell P4 to allocate a comm. buffer */
    p4_recv(&MsgType2, &zero, (char **) &msg, &msg_len);
    num_shifts = *msg;
}
num_nodes = p4_num_total_ids();

worker(id,num_nodes,size,num_shifts);

p4_wait_for_end();
```

The SPMD routine, `worker()`, allocated data for the tests, synchronized all nodes, and then timed the actual ring test. This was followed by the analysis and output of the ring test results:

```
buff_size_bytes = size * sizeof(double);
x = (double*)p4_msg_alloc(buff_size_bytes);
incoming = (double*)p4_msg_alloc(buff_size_bytes);

for(i=0;i<size;i++){
   x[i]   = (double) i;
   incoming[i] = -1.0;
}

p4_global_barrier(type8);

t0 = wtime();
ring(x,incoming,size,num_nodes,num_shifts,id);
ring_time = wtime() - t0;

p4_msg_free(x);            /* Free P4 message buffer  */
p4_msg_free(incoming);     /* Free P4 message buffer  */

min_time = ring_time;
max_time = ring_time;
len      = sizeof(double);

p4_global_op(type7, &min_time, one, len, p4_dbl_min_op, P4DBL);
p4_global_op(type6, &max_time, one, len, p4_dbl_max_op, P4DBL);
```

```
if(id == 0)
     out_results(min_time, max_time, num_nodes, size, num_shifts);
```

The ring test itself was simple. For the NO_SPLIT case, asynchronous sends were posted by each node while blocking receives picked up the messages as they were available.

```
next       = (id +1 )%num_nodes;
prev       = ((id==0)?(num_nodes-1):(id-1));
size_bytes = size * sizeof(double);
len        = size_bytes;

for(i=0;i<num_shifts; i++){
       p4_sendb(type7, next,  x,  size_bytes);
       p4_recv(&type7, &prev, &incoming, &len);
}
```

The analogous code fragment for the SPLIT ring communication pattern follows:

```
if( IS_ODD(id)){
   for(i=0;i<num_shifts; i++){
      p4_sendb(type7, next,  x,        size_bytes);
      p4_recv(&type8, &prev, &incoming, &len);
   }
}
else{
   for(i=0;i<num_shifts; i++){
      p4_recv(&type7, &prev, &incoming, &len);
      p4_sendb(type8, next,  x,        size_bytes);
   }
}
```

While presented as two distinct code fragments in this paper, the two versions were implemented with `#ifdef` macros to assure uniformity outside of the ring communication section itself.

B.3. POSYBL. The POSYBL ring test benchmark consisted of two programs. The first or `master` program setup the benchmark and then became a `worker()`:

```
num_nodes   = atoi(*++argv);
size        = atoi(*++argv);
num_shifts  = atoi(*++argv);

out( lstring("ini:num_nodes"),  lint(num_nodes) );
out( lstring("ini:size"),       lint(size) );
out( lstring("ini:num_shifts"), lint(num_shifts) );
```

```
for( wn= 1; wn < num_nodes; wn++ ) {
    out( lstring("ini:worker"), lint(wn) );
    eval_l( "#5/ring_worker", NULL );
}
worker( 0, num_nodes, size, num_shifts );

out_results( num_nodes, size, num_shifts );

in( lstring("ini:num_nodes"),  qlint(&num_nodes) );
in( lstring("ini:size"),       qlint(&size) );
in( lstring("ini:num_shifts"), qlint(&num_shifts) );
```

The `master` program also analyzed the results by calling `out_results()`. The final set of in's removed the last of the tuples from the tuple manager. Failure to do this results in their still being in tuple space on the next execution of a POSYBL program.

As with PVM and C-Linda, global reduction operations are not part of the system. Hence, we had to create a global minimum and maximum functionality within `out_results()`.

The second or `worker` program just picked up the benchmark parameters from tuple space and then called the `worker()` function.

```
rd(lstring("ini:num_nodes"),  qlint(&num_nodes));
rd(lstring("ini:size"),       qlint(&size));
rd(lstring("ini:num_shifts"), qlint(&num_shifts));
in(lstring("ini:worker"),     qlint(&wn));

worker( wn, num_nodes, size, num_shifts );
```

Both `master` and `worker` call the same SPMD `worker()` function:

```
double *work;   /* work space for global operations */

buff_size_bytes = size * sizeof(double);
x = (double *)malloc(buff_size_bytes);
x = (double *)malloc(sizeof(double));
for(i=0;i<size;i++){
   x[i]   = i;
}

sync(id,num_nodes);   /* synchronize all processes */

t0 = wtime();
```

```
ring(x,size,num_nodes,num_shifts,id);
ring_time = wtime() - t0;

out( lstring("done:time"), lint(id), ldouble(ring_time) );
```

Notice that the POSYBL benchmark differs slightly from the others benchmarks. For POSYBL, the various times are just put into tuple space to be picked up and analyzed by the master (within `out_results()`. This is contrasted with the other programming environments where we used general global reduction functions to analyze the times. The approach used with the POSYBL benchmark was more convenient and had no impact on the timing, so we left this deviation from the normal procedure in place.

A significant difference between POSYBL and C-Linda is the lack of an `inp` operation in POSYBL. This complicated the synchronization operation used before the timing data was collected. Here is the `sync()` function used in the POSYBL benchmark:

```
for ( i = 0 ; i < num_nodes ; i++ )
   out( lstring("sync "), lint(id), lint(i) );

for ( i = 0 ; i < num_nodes ; i++ )
   in( lstring("sync "), qlint(&j), lint(id) );
```

This `sync()` function is very inefficient, but since it falls outside of the timing loop, this would have no impact on the measured times.

The ring communication within the `ring()` function was very simple with POSYBL:

```
/* shift messages around the ring of nodes */

next = (id+1)%num_nodes;
for(i=0;i<num_shifts; i++){
 out(lstring(" shift message"),lint(next),lndouble(x,size));
 in (lstring(" shift message"),lint(id),  qlndouble(x,&len));
}
```

This is analogous to the NO_SPLIT method used for the message passing systems and is essentially identical to the C-Linda code fragment.

B.4. PVM. As mentioned in the text of this paper, we used PVM 2.4 for these tests. Ongoing research will consider PVM 3.X as well, but at the time of these tests, PVM 2.4 was more mature and more efficient.

The PVM benchmark required the development of two distinct executable: one as the host program (`ring_master`) and the other for the nodes (`ring_slave`). Several global entities had to be consistently maintained between these two programs so we needed to use a common include file called `ring.h`. The relevant definitions from this include file are:

```
#define MASTER  "ring_master"  /* name of master executable */
#define SLAVE   "ring_slave"   /* name of slave executable */
#define MsgInit    5           /* type for calc params mesg */
#define MsgRing    1           /* type for ring test mesgs */
#define FALSE      0
#define TRUE       1
```

In addition, we needed to define some additional parameters for use in the PVM program to deal with message types and to support the tracking of executable names (component names).

```
  int is_master;      /* flag set TRUE for master proc only */
  char *next_proc;   /* process name of neighbor */
```

The next code fragment was taken from the host program. Notice that PVM requires program to know and manipulate the names of executables within a computation. The host program takes benchmark data from the command line, enrolls in PVM, initiates the node programs, and then sends the node programs the benchmark data. It then calls the SPMD routine, `worker()`, however, since instance number is not sufficient to identify the node program, we needed to add an additional parameter to `worker()`. This is the first parameter to `worker()` and it serves as a simple flag to notify the SPMD routine of whether the calling program is the host or a node program.

```
  int id;              /* PVM process id number */
  char *proc = MASTER; /* Name of this process */

  num_nodes   = atoi(*++argv);
  size        = atoi(*++argv);
  num_shifts  = atoi(*++argv);

  if ((id = enroll (proc)) < 0) {
    fprintf (stderr, "pvm program: Cannot enroll\n");
    leave ();
    exit (1);
  }

  for(i=1;i<num_nodes;i++) initiate (SLAVE, "SUN4");

  initsend();
  putnint (&size, 1);
  putnint (&num_shifts, 1);
  putnint (&num_nodes, 1);
  snd (SLAVE, -1, MsgInit);   /* -1 destination means broadcast */
```

```
worker(TRUE,id,num_nodes,size,num_shifts);
leave();
```

The node program, enrolls in PVM, fetches computation input, and then calls worker:

```
char *proc = SLAVE;  /* Name of this process */

id = enroll (SLAVE);

rcv (MsgInit);
getnint (&size, 1);
getnint (&num_shifts, 1);
getnint (&num_nodes, 1);

worker(FALSE,id,num_nodes,size,num_shifts);

leave();
exit(0);
```

We now have an SPMD program embodied in the function worker which allocates data for the tests, synchronizes all nodes, and then times the actual ring test. This is followed by the analysis and output of the ring test results:

```
x = (double *)malloc(size * sizeof(double));
incoming = (double *)malloc(size * sizeof(double));

for(i=0;i<size;i++){
   x[i]   = i;
}

i = barrier ("sync",num_nodes);

t0 = wtime();
ring(is_master,x,incoming,size,num_nodes,num_shifts,id);
ring_time = wtime() - t0;
min_time = ring_time;
max_time = ring_time;

gdhi(&max_time,1,&work,num_nodes,id,is_master);
gdlow(&min_time,1,&work,num_nodes,id,is_master);

if(is_master == TRUE)
```

```
    out_results(min_time, max_time, num_nodes, size, num_shifts);
```

As with the C–Linda program, we had to write our own global combine operations to find the maximum and minimum times. As we did not have these from an earlier project, we had to write them specifically for these benchmarks. Given the non-critical timing associated with these routines, we wrote a quick version of these routines in which every node sends its data to the host node where the indicated operation takes place. The host node then broadcasts the results to the other nodes. This is a terrible way to code a global combine operation, but it served our needs and was easy to write and debug. Since global operations are not the purpose of this project, we do not provide code fragments for these operations. Notice that PVM does provide the barrier required for synchronization so global operations aren't completely lacking with PVM.

Finally, the code fragment for the ring portion of the benchmark follows. This version of the ring code is more complex than for C–Linda or P4 since extra logic is required to deal with the fact that each process in the ring is identified by an instance number and a process name.

```
   char *next_proc;   /* process name of neighbor */
   int next;          /* instance number (id) of neighbor */

   if(is_master == TRUE){
     next = 0;            /*  Master always passes on to SLAVE 0 */
     next_proc = SLAVE;
   }
   else{
     next = (id+1)%(num_nodes-1);
     if(next == 0)
       next_proc = MASTER;        /* last SLAVE passes to MASTER */
     else
       next_proc = SLAVE;         /* all other pass to next SLAVE */
   }

      for(i=0;i<num_shifts; i++){
          initsend();
          putndfloat (x, size);
#ifdef SNDRCV
          snd (next_proc, next, MsgRing);
          rcv(MsgRing);
#else
          vsnd (next_proc, next, MsgRing);
          vrcv(MsgRing);
#endif
          getndfloat (in, size);
```

```
      }
```

Notice that the code was set up to investigate both communication mechanisms based on the setting of a compile time switch in the makefile.

The above code fragment corresponds to the NO_SPLIT ring communication pattern. We also considered the SPLIT communication patters which splits the ring of nodes into two sets – one set executing `vsnd/vrcv` and the other set executing `vrcv/vsnd`. The code fragment for this case follows:

```
   if( IS_ODD(id+1)){   /* +1 since two id = 0 in SPMD PVM programs*/
      for(i=0;i<num_shifts; i++){
         initsend();
         putndfloat (x, size);
#ifdef SNDRCV
         snd (next_proc, next, MsgRing);
         rcv(MsgRing);
#else
         vsnd (next_proc, next, MsgRing);
         vrcv(MsgRing);
#endif
         getndfloat (in, size);
      }
   }
   else{
      for(i=0;i<num_shifts; i++){
#ifdef SNDRCV
         rcv(MsgRing);
#else
         vrcv(MsgRing);
#endif
         getndfloat (in, size);
         initsend();
         putndfloat (x, size);
#ifdef SNDRCV
         snd (next_proc, next, MsgRing);
#else
         vsnd (next_proc, next, MsgRing);
#endif
      }
   }
```

We needed to study four cases with PVM since we needed to look at all combinations of communication patterns and message passing routines. While not apparent in these code fragments, the source code selects these alternatives with `#ifdef` macros so we can assure common parts of the program were identical.

B.5. TCGMSG. TCGMSG uses type `long` rather than `int` for its basic message passing parameters. Therefore, the TCGMSG code fragments use their own set of data types.

```
long num_nodes = 0;   /* number of nodes  to use in the test */
long size = 0L;       /* size of vector to use in ring op test */
long type9=9;         /* Message type for tcgmsg */
long type10=10;       /* Message type for tcgmsg */
long id;              /* Node id */
long zero = 0L;
long len;
long num_shifts = 0L; /* number times to shift the message */
```

The TCGMSG program was rigorously an SPMD program. It is only within the `procgroup` file used by the TCGMSG startup program (named `parallel`) that any distinction is made between host and node programs. Therefore, every node used the same top level main program:

```
PBEGIN_(argc, argv);  /* Initialize parallel environment */

num_nodes = NNODES_();
id        = NODEID_();

if (id == 0){
   fp =  fopen("tcgmsg.dat","r"); /* P4 uses command line args   */
   fscanf(fp,"%ld",&size);        /* so we found it easier to    */
   fscanf(fp,"%ld",&num_shifts);  /* load parameters from a file */
   fclose(fp);                    /* rather than the command line*/
}
len = sizeof(long);                          /* Broadcast from  */
BRDCST_(&type9,  &size,       &len, &zero); /* node zero to all*/
BRDCST_(&type10, &num_shifts, &len, &zero); /* other nodes.    */

worker(id,num_nodes,size,num_shifts);

PEND_();                                /* Terminate parallel env. */
```

Notice that the TCGMSG broadcast operation is particularly convenient and does not require distinct `broadcast` and `receive` routines. The worker routine allocates memory, calls the ring test function, and collects and analyzes timing data.

```
int buff_size_bytes; /* length of vector to sum in bytes  */
```

```
buff_size_bytes = size * sizeof(double);
x = (double *)malloc(buff_size_bytes);
incoming = (double *)malloc(buff_size_bytes);

for(i=0;i<size;i++){
   x[i]   = i;
}
SYNCH_(&type10);

t0 = wtime();
ring(x,in,size,num_nodes,num_shifts,id);
ring_time = wtime() - t0;

min_time = ring_time;
max_time = ring_time;

DGOP_(&type8, &min_time, &one, "min");
DGOP_(&type7, &max_time, &one, "max");

if(id == 0)
     out_results(min_time, max_time, num_nodes, size, num_shifts);
```

It is important to note that TCGMSG's synchronous communication forced the programmer to use the far more complicated SPLIT communication pattern in which half the nodes send/receive and the other half receive/send.

```
   long SYNC   = 1;      /* ignored on UNIX workstations */
   long next;            /* node id of the next node */
   long prev;            /* node id of the prev node */
   long nodefrom;        /* actual Node from which message was sent */
   long lenmes;          /* actual length received for a message */

   buff_size_bytes = size * sizeof(double);
   next = (id +1 )%num_nodes;
   prev = ((id==0)?(num_nodes-1):(id-1));

/*
** Since SND/RCV are synchronous, I need to offset the two by
** having the odd id's snd then rcv while even id's do the opposite
*/
   if( IS_ODD(id)){
      for(i=0;i<num_shifts; i++){
```

Table 9
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)

```
          SND_ (&type8, x, &buff_size_bytes, &next, &SYNC);
          RCV_ (&type8, in, &buff_size_bytes, &lenmes,
                &prev, &nodefrom, &SYNC);
      }
   }
   else{
      for(i=0;i<num_shifts; i++){
          RCV_ (&type8, in, &buff_size_bytes, &lenmes,
                &prev, &nodefrom, &SYNC);
          SND_ (&type8, x, &buff_size_bytes, &next, &SYNC);
      }
   }
```

C. Two Node Raw data. In this appendix, we present the raw data from the two node study. 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.

In the course of this investigation, a question about different major releases of C–Linda arose. In Tables 9–10, raw data for two releases of C–Linda (2.4.6 and 2.5) are presented.

PVM 2.4.1 provides two methods for message passing and separates communication from the buffer manipulation. Consequently, PVM benchmarks can vary widely. Tables 13–14 include buffer manipulation while Table 15 excludes it. Note that PVM 3.X programs using the `pvm_advise(PvmRouteDirect)` call should deliver performance analogous to that given in Table 13.

Table 10
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 11
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 12
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)
1,000,000	would not run						

Table 13

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 14

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 15

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 16
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)