Some Experiences with Network LINDA

Document Number G320-????

Draft March 17, 1993

Alan H. Karp

IBM Scientific Center
1530 Page Mill Road
Palo Alto, CA 94304
karp%paloalto@vnet.ibm.com

Abstract

LINDA is a system for programming parallel computers. Although it was originally designed for shared memory machines, versions for distributed memory systems and machines connected over a network are also available. LINDA programs that run well over the network will run well without change on a shared memory system.

This report discusses some features of LINDA as implemented for networked IBM RS/6000s. Measurements of network performance running across both Ethernet and the Serial Link Adapter fiber optic channel are presented. A parallelization of the Linpack 100 algorithm applied to a problem of order 1,000 is described. Some details of how we have configured our system at the Palo Alto Scientific Center that others might find useful are also included.

Index terms: Parallel processing, parallel languages, network computing, RS/6000, LINDA

1 Introduction

Many packages for programming parallel processors are available.[6] For shared memory machines there are language extensions, sets of compiler directives, and preprocessors. For message passing systems there are several library packages, some provided by the hardware vendor and some by third parties.

Shared memory systems view memory as divided into two pieces, a part local to each process and a part shared by all (or some) processes. Thus, each process owns its private memory and shares ownership of the shared memory. Any process can access a word in shared memory simply by referring to it as it would on a sequential machine.

Message passing systems treat the memory of each processor as being accessible only to locally running processes. Data to be shared must be sent explicitly by the programmer. Thus, each process can access only the memory on its processor.

LINDA is unique; it is neither a shared memory nor a distributed memory system. Because of the way it handles memory, LINDA codes can be moved freely between shared and distributed memory machines. What distinguishes LINDA from other approaches is LINDA's handling of shared data objects.

LINDA views memory as divided into two parts. One part is local to each process. The other part, called the tuple (pronounced *toople*) space, is owned by an agent called the tuple space manager. Processes wishing to share data give it to the tuple space manager or request data from it. The tuple space manager keeps track of all the data it has been given and delivers it to requesters. The programmer views the tuple space as a bag containing collections of data that are accessed by name, not address as with conventional memory.

The remainder of this report contains an overview of LINDA, some measurements of Network LINDA running on RS/6000s over both Ethernet and the Serial Link Adapter fiber optic channel, the performance of a parallel version of the Linpack 100 code, and an overview of using LINDA at the Palo Alto Scientific Center. This report ends with a discussion of LINDA's strengths and weaknesses.

2 LINDA Overview

The best overview of LINDA can be found in the reference manual[9] which is also a good primer. Most of the material in this section comes from that document.

The LINDA system from SCA, Inc. consists of several parts. There is a library of user routines for managing tuples, a set of debugging and performance tuning routines, a C-LINDA compiler, a routine for distributing work across the network, and a run-time system that contains the code for the tuple space manager. LINDA also includes a code development system that simulates a parallel processor on a single machine.

Programming in LINDA only requires that you learn how to call six routines – `out`, `in`, `rd`, `inp`, `rdp`, and `eval` which makes writing a parallel program in LINDA rather straightforward. Once you know what data is to be shared among processors, you invoke a function to hand the data to the tuple space manager, `out`. When a process needs some

```
 1:  real_main()                    13:  int worker()
 2:  {                              14:  {
 3:   int i, j, me, np, worker();   15:   int me, j, n;
 4:   np = 5;                       16:   rd("np",n);
 5:   out("np",np);                 17:   in("count",?j);
 6:   out("count",1);               18:   out("count",j+1);
 7:   for ( i=0; i<np; i++)         19:   printf("%d of %d.\n",j,n);
 8:    eval("worker",worker());    20:   return(j);
 9:   for ( i=0; i<np; i++) {      21:  }
10:    in("worker",?j);
11:    printf("%d done.\n",j); }
12:  }
```

Figure 1: Sample program showing one way for each process to get a unique identifier.

data from another process, you use another function to get the data from the tuple space manager.

You have the option of removing the data from tuple space, `in` or `inp`, or simply reading a copy, `rd` or `rdp`. If the data is not in the tuple space, you have the option of blocking until the data appears, `in` or `rd`, or continuing with other work, `inp` or `rdp`.

New processes are started by using `eval` to put a *live* tuple into tuple space. A live tuple starts a new process but does not put anything into tuple space until that process finishes.

This material can be deleted at the editor's discretion.

To make things concrete consider a simple example. We wish to start a number of tasks and have each one compute a unique identifier. The code is shown in Figure 1.

Line 1 shows that the main program must be given the name `real_main`. The first LINDA operation appears in line 5. It says put the string `"np"` and the integer 5 as a single tuple into tuple space. (I think they named it backwards; this operation should be called `in`.) Line 6 initializes the counter to be shared by putting a tuple into tuple space. Lines 7 and 8 show how to start `np` processes by creating live tuples. Each time one of the `worker()` functions finishes, a tuple consisting of the string `"worker"` followed by an integer will be put into tuple space. Finally, the main routine enters into a loop looking for the tuples generated by the `eval`s. If the tuple is not yet in the tuple space, the process will block until the tuple appears.

The worker routine does a `rd` of the number of workers. Next, it does an `in` of the counter and does an `out` of the new value. The `rd` is used for `n` because we want all workers to share the same value; the `in` of `count` is used so that no other worker will grab the value until the update has been completed.

The use of a string for the first entry in the

tuple is more than a nicety. It helps the programmer keep track of what is going on and helps the C-LINDA compiler optimize tuple operations. If we did not have the string, then the tuples would each consist of a single integer. Any tuple space operation that looked just for a single integer could be given this tuple even if we did not want it to. For example, the `in` at line 10 might remove the value of `np` from the tuple space before the workers could read it. The program would then give incorrect results.

Tuples can contain a lot of data, not just single numbers. For example,

```
out("row",i,x,(a+n*i):n)
```

puts a tuple consisting of the string `"row"`, an integer, a floating point number, and an aggregate of `n` floating point numbers into tuple space. If I later need this data,

```
inp("row",7,?x,?b:k)
```

attempts to remove the tuple consisting of the string `"row"`, the integer `7`, any floating point value, and any floating point aggregate. Because I used an `inp` instead of an `in`, I will continue processing if the tuple is not yet in tuple space. The function `inp` returns a value of 1 if successful and 0 otherwise. If the `inp` was successful, `k` is set to the number of items returned. Due to a bug in the current release, `k` will be set to a core constant if the `inp` fails.

It is also possible to use anonymous fields in LINDA calls. Let's say I want to remove all the tuples beginning with `"row"` but don't need the data. The call

```
inp("row",?int,?float,?float *:)
```

will remove the tuple without requiring me to allocate space for the data. This feature is particularly important when running across the network because data transmission takes so long. No data is transferred for anonymous fields.

End of optional material.

In addition to the network version, the LINDA system has a code development system (CDS). The CDS allows you to simulate parallel processing on a single machine. It is useful for testing and debugging codes. First of all, you do not have to wait for executables to be moved to several machines before your run starts. Secondly, you can simulate many more processors than are available for runs. Finally, the code development system comes with the tuple scope, an X-windows based tool that lets you view and interact with the tuple space during your run.

The tuple scope displays a *pane* for each class of tuples you define in your program. You may run, stop, or single step your program. You may also examine tuples or dump the tuple space. There is also a debug language to help you see what is going on. Due to a bug in the X interface, nothing will happen until you click on any pane's icon and then move that pane.

This brief introduction is intended only to give a flavor for the LINDA system. Further details can be found in the C-LINDA Reference Manual[9], a book by LINDA's inventors[2], and a review article[1].

3 LINDA at PASC

The Parallel Processing Laboratory (PPL) at the Palo Alto Scientific Center (PASC) is in

a constant state of flux. As new equipment becomes available it is added to the machines. Thus, the configurations given here are likely to be out of date. In fact, the configuration changed while this paper was being written necessitating changes to all the figures! While much of the material in this section is specific to our configuration, others may be able to build on our experience when configuring their systems.

There are 5 machines in the PPL; all are on the Ethernet, and 3 are connected in a line with Serial Link Adapter (SLA) fiber optic channels. Figure 2 shows the system configuration. All machines on the fiber optic network are model 530s. We found that there were problems when a model 530 sent data to a model 520 over the SLA. Apparently, the 520 could not keep up with the data flow.

After some thrashing around, we found a reasonable way to manage these machines. In particular, we were able to deal with multiple networks connected to each machine. We also found a convenient way to switch between the code development system and the network version.

Each of the machines on the SLAnet has a different name and IP address for each network to which it is connected, in our case two. The name itself is used to communicate over the Ethernet; the name with `-fo` (for fiber optic) appended uses the SLA channel. The network used will also depend on the hostname. If `knack` has its hostname set to `knack-fo`, and it sends a message to `knick-fo`, then all transmission is over the SLAnet. If `knack` has hostname `knack` and talks to `knick`, everything goes over the Ethernet. If `knack` has hostname `knack` and talks to `knick-fo`, the transmission from `knack` uses the SLA but the acknowledgements use the Ethernet.

There is one other consideration when looking at `hostname`. In order to use LINDA to run across the network, the user provides a file `tsnet.nodes` listing the machines to be used. LINDA checks that the value returned from `hostname` is the same as the entry in this file. If you want to use the SLAnet, you must specify the `-fo` form of the machine names and make sure that the host name is set properly. We have been told that future releases will not check `hostname` so this problem is temporary. In the interim, your installation should set up procedures to allow your users to change the `hostname`.

Originally, we ran without any sort of network file system. Users had to move their code and data to all machines to be used before the run started, a process automated by the LINDA system. While the 5 to 10 second delay was not a problem for production runs, it was an annoyance when making short test runs. Things became more convenient once we brought up the Andrew File System (AFS).[8]

We found that interactive use and the load due to AFS interfered with our parallel runs. This problem was minimized when we limited the use of our three machines on the SLA network to parallel use only. Thus, all LINDA runs were started from a machine not on the fiber optic network. While some data transfers were slower, the problem was not too serious because we were able to structure our algorithms so that all our large data aggregates were generated on the machines doing the computation.

The shell scripts we received from SCA

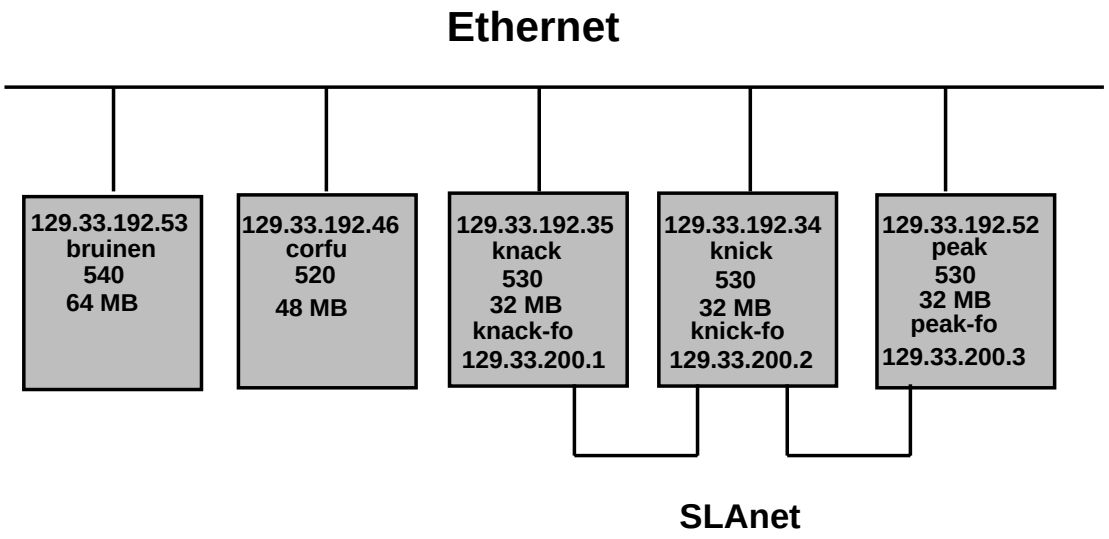

Figure 2: Configuration of the PPL as of 1 July 1991. The long numbers are the IP addresses for the two networks. There is no connection between `peak-fo` and `knick-fo` because one of our fiber optic boards has a defective side.

did not make it easy to switch between the network and code development versions. To improve the situation we modified the scripts and installed LINDA in a directory with two subdirectories – `tsnet` for the network code and `cds` for the code development system. Users now include both the `cds/bin` and `tsnet/bin` directories in their path. They can run with the version they want simply by setting the environment variable `LINDA` to either `cds` or `tsnet`.

When the environment variable `LINDA` is set to `tsnet` the job will run in parallel using the machines listed in the file `tsnet.nodes`. Any machine that allows you to do a remote copy to it can be used so you can grab cycles from others. We limited our work to the three dedicated machines in the PPL so as not to irritate our colleagues.

Unless you tell the system how many processors you wish to use, LINDA will distribute your executable to all nodes in your `tsnet.nodes` file unless you use a network file system like NFS or AFS or distribute the code ahead of time. This procedure can take quite a while if you list a lot of machines.

There are two ways to avoid this delay. If you specify `tsnet -n x`, LINDA will attempt to find the `x` least heavily loaded machines in your list; you won't know which machines LINDA will pick. If it is important to pick the machines you use, you will have to modify your `tsnet.nodes` file to list only the machines you wish to use.

Even though only C-LINDA is available, you may still use Fortran for the bulk of your

code. You will have to write some simple C to handle the tuple space operations, though. Appendix E is the set of C routines needed to run the parallel Linpack code discussed in Section 5.

4 Network Performance

LINDA manages communications for the user. This feature makes it easier to get a program running, but it also makes it hard to interpret the network performance measurements. In this section I present some such measurements along with an attempt at interpreting them.

The `ping` time tells us the minimum transmission time between two machines. It takes about 3 ms to ping any two machines over the Ethernet, 2 ms between neighbors on the SLAnet, and 4 ms on the SLAnet with one intervening node.

LINDA does its best to optimize the tuple space operations. In a network environment the tuple space manager is distributed across all processors which avoids a potential serial bottleneck. The tuples are distributed among these tuple managers.

All measurements presented in this section were made using the code in Appendix A and the results summarized in Table 1. It times the `out` of tuples of various sizes, then the `in` of them. This procedure is repeated to test for initialization effects such as memory allocations. (When run on a single machine, the two `eval`s of `indata()` are changed to function calls.)

First, let's look at LINDA running on a single RS/6000-530, `peak-peak` in Table 1. Some special cases must be considered. For example, LINDA does dynamic memory allocation to make room for new tuples. There is, therefore, a difference between the performance of the first iteration and subsequent ones. In addition, tuples larger than 8 KB are handled differently than smaller ones. Single quantities and small aggregates are put directly into tuple space; large aggregates are copied to a local memory buffer, and a pointer to this data is put into tuple space.

An `out` or `in` of a small tuple takes about 0.4 ms which is a measure of the software overhead. As shown in Figure 3, the access time does not start to increase until the tuples are about 1 KB long. Note that the the first `out` of large aggregates is slower than subsequent `out`s due to the time needed for memory allocation. The `in` times are nearly the same as the times for the second `out`. The times on a model 520 are longer by a factor of about 1.4. Small variations are not significant and can be attributed to various Unix demons stealing cycles.

Next, let's look at two RS/6000s connected with an Ethernet or an SLAnet, `peak` to `knick` and `peak-fo` to `knick-fo`, respectively. Getting a small tuple from another machine takes about 5.3 ms over the Ethernet and 4.3 ms over the SLAnet, only about twice the `ping` time. Also, the performance for an `out` of a 1 MByte tuple is nearly the same as when running on one processor. This equality is due to LINDA keeping large aggregates in a local memory buffer.

The performance for `in`s is shown in Figure 4. We see that there is only a small difference between the performance of a 520 talking to a 530 and a 530 talking to a 530.

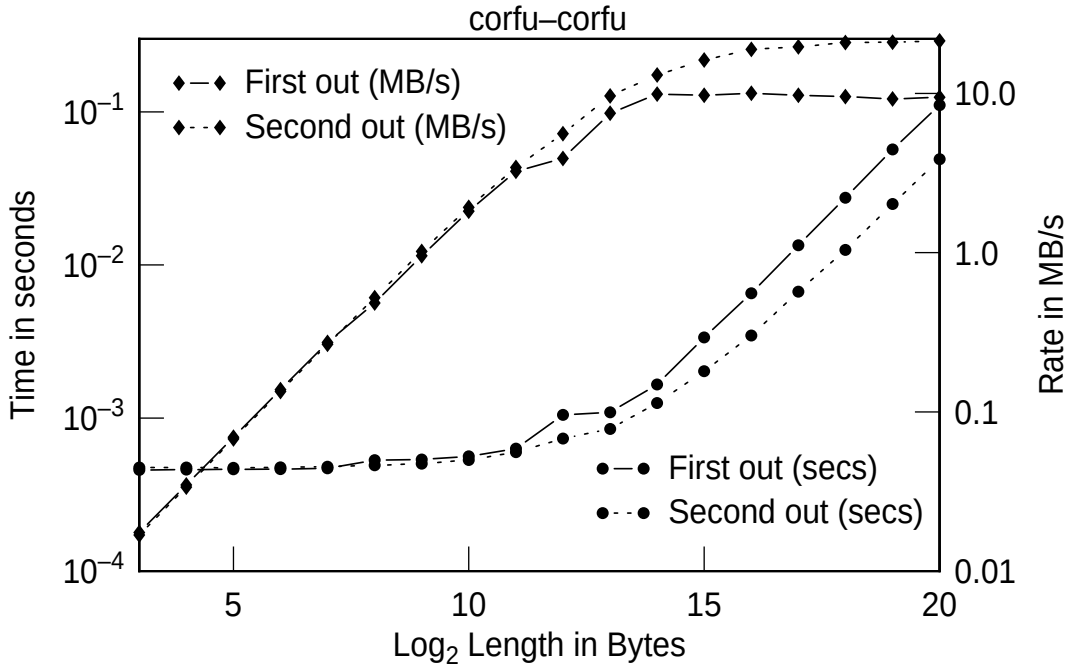

Figure 3: Times and rates to `out` tuples of various sizes on a model 520. The solid line shows the times for the first `out`; the dashed line, the second `out`.

There is a large difference between the performance of nearest neighbors and machines separated by an intermediary. (Recall that `knick-fo` sits between `peak-fo` and `knack-fo` on the SLAnet.) The fiber optic board can forward messages without interrupting the CPU if we use low level calls. However, if we use IP, as LINDA does, the CPU is needed to provide address resolution.

LINDA distributes tuples and small aggregates among the machines. For example, it takes 5.3 ms to get a 16 byte tuple from `peak` to `knick` but only 0.4 ms for an 8 byte tuple. This effect can be attributed to the smaller tuple being stored on the machine that ultimately does the `in`. Further evidence is the fact that the `out` of the 8 byte tuple takes about 1 ms compared to 0.4 ms to `out` the 16 byte tuple.

Notice the change in performance between aggregates of 8 KB and those of 16 KB shown in Table 1. Large aggregates are held in a memory buffer on the machine that does the `out` and a pointer to this buffer is put into tuple space. The time to `out` the smaller tuple is 1.6 ms compared to 0.85 ms to `out` the larger tuple. In fact, the time to `out` any tuple larger than 8 KB is the same as when run on one machine. The situation for the `in`s is more complicated, but the change in performance for 8 KB tuples shows that there is a difference between the handling of large and small tuples.

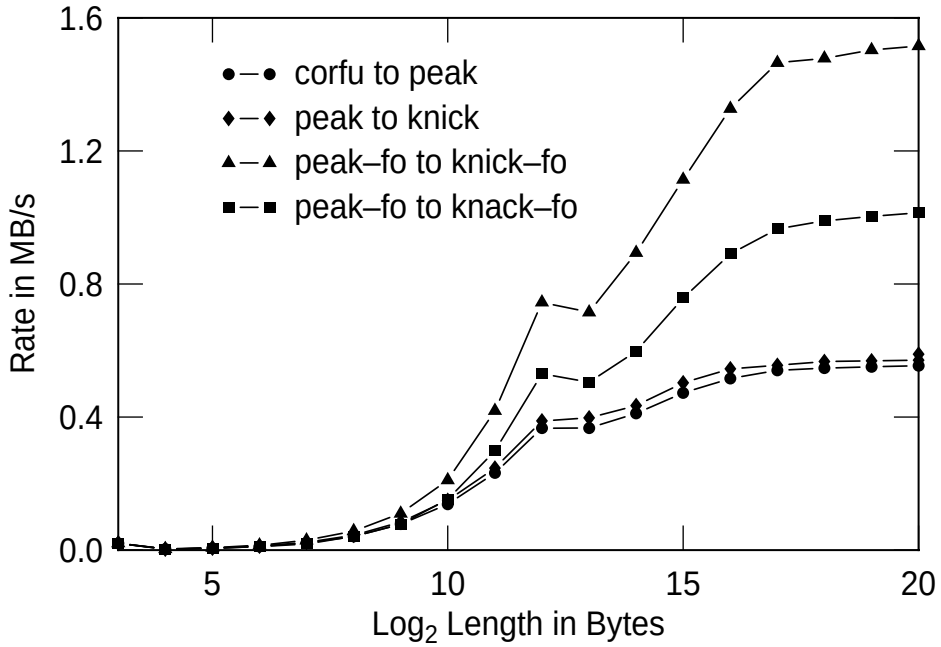

Figure 4: Times to `in` tuples of various sizes over the network. `corfu` is a model 520; `peak`, `knick`, and `knack` are model 530s. The `-fo` suffix means that the fiber optic link was used.

This performance for the SLA, nominally a 20 MB/sec link, is disappointing especially considering the fact that LINDA uses UDP instead of the less efficient TCP for communications. The next release of LINDA should improve the performance substantially. The current release is using an 8 KB buffer which limits performance to a theoretical peak of 1.6 MB/sec. The new release will use a 28 KB buffer which should result in transfer rates exceeding 6 MB/sec.

While I had the machines to myself, I ran some other tests which are summarized in Figure 5. Clearly, `peak` to `knick-fo` uses the SLAnet but is a few percent slower than when both names use the `-fo` suffix. On the other hand `peak-fo` to `knick` is using the Ethernet but is slightly faster than when neither name has the `-fo` suffix.

The three processor run is using the SLAnet to communicate between `peak-fo` and `knack-fo`. Another iteration made as part of this run had `peak-fo` talking to `knick-fo`. The measured rate to `in` a tuple of 4 KB was over 7 MB/s while the rate for an 8 KB tuple was 0.54 MB/s. The run made entirely on `peak` also showed 7 MB/s transfer rate. Clearly, the 4 KB tuple created by `peak-fo` was being stored on `knick-fo`. Very large aggregates were transferred at 1.5 MB/s between nearest neighbors and at 1.0 MB/s with one intervening

Machines	Op	8B	16B	4 KB	8 KB	16 KB	1 MB
corfu	out	0.000458	0.000460	0.001049	0.001091	0.001656	0.110827
	out	0.000475	0.000475	0.000734	0.000849	0.001255	0.049091
corfu	in	0.000509	0.000508	0.000768	0.001039	0.001456	0.050214
peak	out	0.000357	0.000360	0.000784	0.000805	0.001159	0.075027
	out	0.000359	0.000359	0.000547	0.000625	0.000844	0.032070
peak	in	0.000392	0.000391	0.000581	0.000751	0.000967	0.032262
peak	out	0.001071	0.000496	0.000790	0.001533	0.001166	0.078526
	out	0.001068	0.000377	0.000564	0.001629	0.000862	0.032279
knick	in	0.000384	0.005297	0.010544	0.020618	0.037712	1.835715
peak-fo	out	0.001331	0.000485	0.000785	0.001977	0.001163	0.077335
	out	0.001329	0.000365	0.000547	0.001638	0.000846	0.032120
knick-fo	in	0.000384	0.004292	0.005583	0.011707	0.018338	0.681707
peak-fo	out	0.001356	0.000489	0.000913	0.001930	0.001153	0.076972
	out	0.001327	0.000365	0.000549	0.001648	0.000846	0.032575
knack-fo	in	0.000393	0.005947	0.008215	0.015510	0.027634	1.034006
corfu	out	0.001835	0.000611	0.000625	0.001038	0.002316	0.111513
	out	0.001369	0.000473	0.000725	0.002124	0.001256	0.050180
peak	in	0.000390	0.005779	0.011182	0.022316	0.039868	1.890971
corfu	out	0.001992	0.000597	0.001041	0.002047	0.002157	0.110479
	out	0.001335	0.000462	0.000713	0.001797	0.001270	0.048965
peak-fo	in	0.000384	0.005730	0.011156	0.022147	0.040049	1.884350
peak	out	0.001357	0.000493	0.000788	0.001821	0.001157	0.077085
	out	0.001325	0.000366	0.000550	0.001633	0.000847	0.032092
knick-fo	in	0.000382	0.005095	0.006068	0.012111	0.020148	0.738569
peak-fo	out	0.001369	0.001520	0.001745	0.001606	0.001278	0.078623
	out	0.001347	0.001358	0.001583	0.001640	0.000853	0.033222
knick-fo	in	0.000387	0.000382	0.000572	0.015102	0.018865	0.695202
knack-fo	in	0.004408	0.004411	0.005829	0.015684	0.027171	1.041857

Table 1: Summary of measurements of LINDA network performance. All times are in secconds.

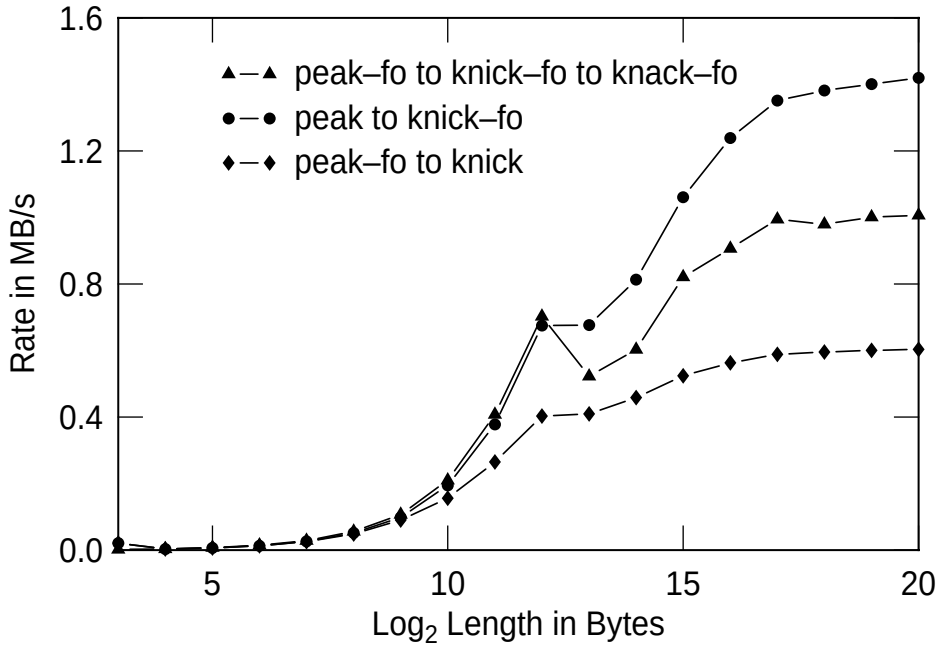

Figure 5: Some timings of `in` using different combinations of networks.

node.

Selected times for the runs reported in this section are given in Table 1. Some aspects of these times are worth noting. On the two processor runs, the `out` times are small for the 16 B tuples and large for the 8 B tuples; the roles are reversed for the `in` times. The three processor run shows a similar effect, but here all small aggregates tuples are stored on `knick-fo`. Larger tuples are always sent across the network.

5 Parallel Linpack 100

One way to learn a new programming system is to rewrite a program you know extremely well. This approach lets you concentrate on what is new without having to worry about not understanding fully the problem being solved. I chose to take the Linpack 100 algorithm run on a problem of order 1,000 as my sample.

The original code is shown in Appendix B. This subroutine factors a matrix `A` into lower and upper triangular parts using Gaussian elimination. These factors over-write the input matrix. At each step, the current column is searched for its largest element, and the row containing this element exchanged with the row containing the diagonal element, a process called pivoting. Next, a set of multipliers are computed and applied to the remaining submatrix. This procedure is repeated on the submatrix until the submatrix is a single ele-

ment.

This code is not a good one for parallel processing. It is based on the BLAS 1,[7] a vector-vector approach, which does not make effective use of hierarchical memories. Since the distributed memory of the parallel machine can be viewed as another stage in the hierarchy, we would expect that an algorithm based on the BLAS 3[4] would perform much better. Indeed, on a single RS/6000 model 520 this method runs at only 9 Mflops/sec on an order 1,000 problem while an implementation based on the BLAS 3 runs at 26 Mflops/sec.[3]

The first decision made in parallelizing this code is the data distribution – rows or columns, block or wrapped. The algorithm is column oriented since Fortran stores data in column major order. I decided to distribute complete columns to each processor to avoid communication during the pivot search. Thus, pivoting is guaranteed to take a trivial amount of time. Also, wrapped distribution in which columns are dealt to processors like cards in a bridge game provides better performance than giving each processor a contiguous block of columns. (The best approach is a hybrid – deal out contiguous blocks of columns – but was too complicated for this learning exercise.)

The program, shown in Appendix C, is nearly 4 pages long, compared to the orginal 25 lines of executable code. Clearly, LINDA is more suited to a program with large blocks of serial code where the effort of parallelizing per line of code is more modest.

The first thing the program does is start he workers. Next, it puts the array `A` into tuple space one column at a time. As soon as the workers are started, they begin removing the columns from tuple space so there is some overlap between the input and output processes. Once all the columns have been put into tuple space, the master enters a loop looking for results. Each time a column has been updated, the owning process puts it into the tuple space so the input of the results can overlap the computation.

This output loop also removes the pivots from the tuple space as soon as it is safe to do so. If this step were neglected, the next call to this routine would give incorrect answers because a `rd` of a pivot column could get one left from a previous matrix. For neatness sake, the master task also removes the tuples left by the `eval`s.

The routine `control` simply inputs the columns and calls the routine that updates the matrix. You will notice that this implementation could lead to super-linear speed-ups. If the matrix is too large to fit in the memory of one of the processors but the individual pieces do fit, the one processor run will be slow due to paging while the multiprocessor runs will work entirely out of memory.

Upon entering routine `update`, the owner of column 1 pivots the column and `out`s the first set of multipliers and the first column of the results. Everyone else attempts to `rd` the first pivot column.

The owner of the next column immediately generates the next pivot column so it will be ready when the others have finished updating their parts of the remaining submatrix. It also `out`s the next column of the results. All others proceed to the loop that updates the submatrix.

We can estimate the performance of this implementation using the sequential program for the compute time and our measurements of network performance for the communications

time. All estimates are for an order 1,000 problem running on a model 530. The serial computation runs at over 12 Mflops/sec and takes about 54 seconds.

The master takes 2 ms to `out` each column, a total of 2 seconds. Since the master is not on the fiber network, getting the data from the master to the worker proceeds at a rate of 0.37 MB/sec and takes 22 ms per column or 22 seconds for the entire matrix. Since each processor `in`s only its part of the data, as a rough estimate we can divide this time by the number of processors. Of course, this estimate is too optimistic since eventually the transfer rate stops improving as either the network, the tuple space manager, or both become overloaded.

The time to `out` the pivots also takes about 2 seconds, but this figure can be divided by the number of processors since each machine must output only part of the result. The time to get the pivots is a bit more complicated to estimate since each has a different length; the first is length n; the next, length $n-1$; *etc*. If we let the data transmission time be

$$t_k = t_0 + l_k/r,$$

where t_0 is the start-up time, l_k the length of k'th pivot column, and r the transfer rate, the total time is

$$t_t = \sum_{k=1}^{n-1} t_k = (n-1)t_0 + \frac{4(n+2)(n-1)}{r}.$$

We have $r = 1.5$ MB/sec, $t_0 = 0.004$ sec so that $t_t = 7$ sec.

Each processor will also `out` its part of the results into tuple space so they can be collected by the master, a total of 2 second. The master must collect the results which will take 22 seconds.

The total time will be the sum of all these times. If we use p processors, our time to complete the calculation will be

$$T = 24 + 88/p.$$

We have assumed no overlap between input, output, and computation.

Of course, this estimate is too optimistic because the data transfer rates do not decrease linearly with p. Our actual times are somewhat longer than predicted. For $p = 2$ we predict a time of 68 seconds and measure 73 seconds; for $p = 3$ we predict 53 seconds and measure 67.

It is clear that a good place to look for improvement is in getting the data from the master to the workers. Not only is it by far the largest part of the communications time, but a simple change can be made to improve the transfer rate by about 40%. From our network performance measurements we see that 8 KB blocks of data are transferred at a rate of 0.3 MB/sec while 128 KB blocks move at over 0.5 MB/sec. Thus, the output routine was modified to use a work array of size 16 K words to increase the size of the aggregates. The routines modified to implement this change are shown in Appendix D.

With this change we expect it to take 15 seconds to load the columns into a single processor; we measure 16 seconds. The overall performance improved a bit less than expected due to the loss of some overlap between the `out`s and `in`s of the columns. Table 2 compares the performance of these two ap-

Machines	Method	out col	in col	Update	in res	Total	Mflops
A	Unblocked	4.84	32.75	77.88	107.7	113.6	5.87
A,B	Unblocked	12.48	18.12	46.66	60.40	73.05	9.12
A,B,C	Unblocked	14.89	18.52	43.39	51.64	66.61	10.00
D,E	Unblocked	13.24	18.21	48.35	57.51	70.89	9.40
X	Unblocked	7.01	14.02	56.54	63.85	70.98	9.40
A	Blocked	0.87	16.45	78.34	97.41	98.31	6.78
A,B	Blocked	2.06	11.33	45.02	62.38	64.52	10.32
A,B,C	Blocked	4.66	9.09	41.36	52.82	57.65	11.55
D,E	Blocked	2.15	9.69	48.63	62.92	65.23	10.21
X	Blocked	1.18	5.20	53.27	58.13	59.37	11.23

Table 2: Summary of Linpack 100 algorithm on a problem of order 1,000. A – `peak-fo`, B – `knick-fo`, C – `knack-fo`, D – `peak`, E – `knick`. In all cases except X `corfu` is the master processor; in X it is `peak-fo` with `knick-fo` as the worker. The best of four runs is shown. All entries but Mflops are in seconds.

proaches. Each table entry is the best of four runs on a random matrix.

Some features of these results are worth noting. The total time is very nearly equal to the time needed to `out` the columns plus the time to `in` the results. The total is slightly larger than the sum for some unknown reason.

We would also expect the time to `in` the columns plus the time to update the submatrices to equal the time to `in` the results. We see that for the unblocked versions the sum is greater than the time to `in` the results. Clearly, we are overlapping the time to `out` the columns and `in` the results with the time to do the update.

We have lost much of this overlap with the blocked code; the sum of the time to `in` the columns and update the array is substantially less than the time to `in` the results. Clearly, `corfu` can not get the results across the Ethernet fast enough one column at a time to keep up with the computation. Only when all processors are connected over the SLAnet is the sum close to the time to `in` the results.

I made another set of runs to see the effect of having more machines listed in the `tsnet.nodes` file than I had tasks. With `corfu` as the master, I listed `peak`, `knick`, and `knack` and specified 2 tasks. Each pass ran on a different pair of processors due to the way LINDA assigns `eval`s to processors. The processor with no computational work to do had a CPU busy of 10% to 20% and little I/O activity except for very short bursts. Overall performance was reduced from 9.4 Mflops/sec when two processors were in the `tsnet.nodes` file to 9.0 Mflops/sec. Clearly, there is some extra communications overhead when the third processor is given tuples by the tuple space manager.

6 Conclusions

LINDA does what it says it will do and does it well. The code is stable, does not crash the system, has only minor bugs, and uses the network efficiently. There are a number of annoying errors in the documentation, however, that make using some features frustrating. For example, the tuple scope can be set up to stop the program when some condition is met. The documentation lists "=" as the test for equality. Unfortunately, equality is tested with "==". Another problem was a missing underscore in the names of the timer functions. In both cases, e-mail support quickly identified the problem.

The network performance over the SLA is disappointing but is supposed to be improved by about a factor of four in the next release. Even with this improvement network LINDA, like other network computing systems, is only suitable for large grain work. In the 6 ms it takes to get a small tuple from another machine you could have executed 60,000 floating point operations. Even accessing a local tuple consumes the time needed to do 3,000 flops. If you can structure your problem to transfer large aggregates, things become somewhat better. At 1.5 MB/sec you can do only about 60 flops in the time it takes to transfer a double precision word.

Although LINDA has been used for a wide variety of applications,[5] I believe it would be much more useful if some features were added.

It would be nice to have some kind of control over the tuple space as a whole. For example, the parallel Linpack code would have benefited from a `clear_ts` function that would simple empty the tuple space. It takes over 6 seconds to remove the 1,000 pivot tuples, a function that could be done with a single call.

Library routine writers would also benefit if they could push and pop tuple spaces. It is possible to achieve a similar function now by getting a random number when the routine is entered and using it as an actual in each tuple operation. However, getting a unique sequence of random numbers for disjoint processes is not trivial. Alternatively, we could use a tuple to keep track of the push/pop level. While it is not hard to implement, this scheme needs care since the library routine does not get access to tuple space until some unknown amount of work has been done. A pair of `push_ts` and `pop_ts` calls would be simpler.

As presently implemented, there is no way to control where a tuple goes. This feature makes programming easier but means that I can't stage data to the machine that will need it later. It would be nice if I could overlap the network delay with computation.

A simple change in the interpretation of `rdp` and `inp` would allow me to overlap data movement and computation. Currently, these functions return a 0 and do no data movement if the tuple is not in tuple space. I propose they also return 0 if the data is not on the requesting processor. However, in this case the data will be moved so at a later time the operation will return a 1 and move the data from a local buffer to my array. This change would allow me to issue the `rdp` or `inp` as soon as I know what data I want. A later `rd` or `in` will provide any needed synchronization.

Many programs could be greatly simplified if the tuple manager had some intelligence.

For example, if I could

```
in("x",?k<7)
```

which returns any tuple for which k is less than 7, I could reduce both the number of synchronization points in the code and the number of accesses to tuple space. Of course, any valid function should be acceptable.

References

[1] Nicholas Carriero and David Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[2] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs: A First Course*. The MIT Press, Cambridge, MA, 1990.

[3] Jack J. Dongarra. Performance of Various Computers Using Standard Linear Equation Software in a Fortran Environment. Technical Report CS-89-85, University of Tennessee, March 1990.

[4] Jack J. Dongarra, J. DuCroz, Ian Duff, and Sven Hammerling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16:1–17, 1990.

[5] David Gelernter. Getting the Job Done. *Byte*, page 301, November 1988.

[6] Alan H. Karp. Programming for Parallelism. *IEEE Computer*, pages 43–57, May 1987.

[7] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5:308–329, 1979.

[8] Mahadev Satyanarayanan. Scalable, Secure, and Highly Available Distributed File Access. *IEEE Computer*, 23(5):9–21, May 1990.

[9] Andrew H. Sherman. *C-Linda Reference Manual*. New Haven, CT, 1990.

A Network Measurement Code

```
/* Time network performance of C-Linda */
#include <time.h>
#include <stdio.h>
#define SIZE 524288
#define MAX 18
#define TIMES 5

real_main()
{
   int outdata(), indata();
   ("outdata",outdata());
   eval("live",indata());
   in("live",?int);
   ("outdata",outdata());
   eval("live",indata());
}

int outdata()
{
   int i, j, l, it, indata();
   double *a, mint, t;
   setlinebuf(stdout);
   a = (double *) malloc(SIZE*sizeof(double));
/* Time outs */
   printf("Time out on varying sizes on\n");
   system("hostname");
   l = -1;
   printf("Time for single integer: ");
   for (i=0;i<TIMES;i++) {
       it = mytime();
       out("data",l);
       printf("%10.6f ",1e-6*(mytime()-it));
   }
   printf("\n");
   for (j=0;j<MAX;j++) {
       l = 1<<j;
       mint = 1e20;
       printf("%10d",j+3);
       for (i=0;i<TIMES;i++) {
           it = mytime();
           out("data",l,a:l);
           t = 1e-6*(mytime() - it);
```

```
            printf("%10.6f ",t);
            if ( t > 0.0 )
               mint = (mint<t)?mint:t;
        }
        printf(" %10.6f %10.6f\n",mint,1e-6*8*l/mint);
    }
    return(0);
}

/* Now time ins */
int indata()
{
    int i, j, l, it;
    double *a, mint, t;
    setlinebuf(stdout);
    a = (double *) malloc(SIZE*sizeof(double));
    printf("Time in on varying sizes on\n");
    system("hostname");
    l = -1;
    printf("Time for single integer: ");
    for (i=0;i<TIMES;i++) {
        it = mytime();
        in("data",?l);
        printf("%10.6f ",1e-6*(mytime()-it));
    }
    printf("\n");
    for (j=0;j<MAX;j++) {
        l = 1<<j;
        mint = 1e20;
        printf("%10d",j+3);
        for (i=0;i<TIMES;i++) {
            it = mytime();
            in("data",l,?a:);
            t = 1e-6*(mytime() - it);
            printf("%10.6f ",t);
            if ( t > 0.0 )
               mint = (mint<t)?mint:t;
        }
        printf(" %10.6f %10.6f\n",mint,1e-6*8*l/mint);
    }
    return(0);
}
```

B Original Linpack 100 Code

```
      SUBROUTINE DGEFA(A,LDA,N,IPVT,INFO)
      INTEGER LDA,N,IPVT(1),INFO
      DOUBLE PRECISION A(LDA,1)
      DOUBLE PRECISION T
      INTEGER IDAMAX,J,K,KP1,L,NM1
C
C     GAUSSIAN ELIMINATION WITH PARTIAL PIVOTING
C
      INFO = 0
      NM1 = N - 1
      IF (NM1 .LT. 1) GO TO 70
      DO 60 K = 1, NM1
         KP1 = K + 1
C
C        FIND L = PIVOT INDEX
C
         L = IDAMAX(N-K+1,A(K,K),1) + K - 1
         IPVT(K) = L
C
C        ZERO PIVOT IMPLIES THIS COLUMN ALREADY TRIANGULARIZED
C
         IF (A(L,K) .EQ. 0.0D0) GO TO 40
C
C           INTERCHANGE IF NECESSARY
C
            IF (L .EQ. K) GO TO 10
               T = A(L,K)
               A(L,K) = A(K,K)
               A(K,K) = T
   10       CONTINUE
C
C           COMPUTE MULTIPLIERS
C
            T = -1.0D0/A(K,K)
            CALL DSCAL(N-K,T,A(K+1,K),1)
C
C           ROW ELIMINATION WITH COLUMN INDEXING
C
            DO 30 J = KP1, N
               T = A(L,J)
               IF (L .EQ. K) GO TO 20
                  A(L,J) = A(K,J)
```

```
                  A(K,J) = T
   20          CONTINUE
               CALL DAXPY(N-K,T,A(K+1,K),1,A(K+1,J),1)
   30       CONTINUE
         GO TO 50
   40    CONTINUE
            INFO = K
   50    CONTINUE
   60 CONTINUE
   70 CONTINUE
      IPVT(N) = N
      IF (A(N,N) .EQ. 0.0D0) INFO = N
      RETURN
      END
```

C Parallel Linpack 100 Code

```
      SUBROUTINE DGEFA(A,LDA,N,IPVT,INFO)
      INTEGER LDA,N,IPVT(1),INFO
      DOUBLE PRECISION A(LDA,1)
      DOUBLE PRECISION T
      INTEGER IDAMAX,J,K,KP1,L,NM1
C
C     GAUSSIAN ELIMINATION WITH PARTIAL PIVOTING
C
      data if/0/
      if ( if .eq. 0 ) then
         if = 1
         read(1,*) nt
         write(*,*)nt,' tasks.'
         close(1)
         if ( nt .eq. 0 ) call exit
      endif
      INFO = 0
      IF (N-1 .ge. 1) then
         call init(%val(n),%val(nt))
         do i = 1, nt
c---------> eval("evals",ccontrol(i,nt,n))
            call spawn(i,nt,n)
         enddo
         t1 = second()
         do j = 1, n
c---------> out("col",j,a(1,j):n)
```

```
            call outcol(n,j,a(1,j))
         enddo
         write(*,*)second()-t1,' seconds to output columns'
         t1 = second()
         do k = 1, n + nt
c------------------------> in("result",k,?ipvt(k),?a(1,k):n)
            if ( k .le. n ) ipvt(k) = inres(k,l,a(1,k))
c------------------------> in("pivot",k-nt,?int,?int,?double,?double *:)
            if ( k .gt. nt ) call inpiv ( k - nt )
         enddo
         write(*,*)second()-t1,' seconds to input results'
c-----------> in("evals",?int)
         call cleanup(nt,n)
      endif
      end
c---------------------------------------------------------------------
      function control(me,nt,n,aj,ak)
      double precision aj(n,*), ak(n), t
      mycol(k) = (k+nt-1)/nt
      kreal(k) = me + nt*(k-1)
C
C           ROW ELIMINATION WITH COLUMN INDEXING
C
      control = 0.0
c
c Read in my subset of the columns
c
      ncol = mycol(n)
      t1 = second()
      do i = 1, ncol
         k = kreal(i)
         if ( k .le. n ) then
c-------------> in("col",k,?aj(1,i):n)
            call incol(n,k,aj(1,i))
         endif
      enddo
      write(*,*)me,': ',second()-t1,' seconds to read columns'
      call update(me,nt,n,aj,ak)
      RETURN
      END
c---------------------------------------------------------------------
      subroutine update(me,nt,n,aj,ak)
      double precision aj(n,*), ak(n), t
      mycol(k) = (k+nt-1)/nt
```

```
      metest(k) = 1 + mod(k-1,nt)
      kreal(k) = me + nt*(k-1)
c
c Loop over cycle
c
      t3 = second()
      if ( me .eq. 1 ) call pivcol(nt,1,n,aj)
      do k = 1, n - 1
c-----------> rd("pivot",k,?l,?t,?ak:(n-k+1))
         call rdpiv(k,l,t,ak)
c
c Update the next column and get the pivot out ASAP
c
         if ( metest(k+1) .eq. me ) then
            j = mycol(k+1)
            T = Aj(L,j)
            IF (L .ne. K) then
               Aj(L,j) = Aj(K,j)
               Aj(K,j) = T
            endif
            CALL DAXPY(N-K,T,Ak(K+1),1,Aj(K+1,j),1)
            call pivcol(nt,k+1,n,aj(1,mycol(k+1)))
         endif
c
c Update my columns using the given pivot
c
         do j = mycol(k+2), mycol(n)
            if ( j .le. n ) then
               T = Aj(L,j)
               IF (L .ne. K) then
                  Aj(L,j) = Aj(K,j)
                  Aj(K,j) = T
               endif
               CALL DAXPY(N-K,T,Ak(K+1),1,Aj(K+1,j),1)
            endif
         enddo
      enddo
      tot = second() - t3
      write(*,'(a,i5,a,f12.6)')'update:',me,': total time =',tot
c
      end
c-----------------------------------------------------------------
      subroutine pivcol(nt,k,n,a)
      double precision a(n), t
```

```
C
C        FIND L = PIVOT INDEX
C
      L = IDAMAX(N-K+1,A(K),1) + K - 1
C
C     ZERO PIVOT IMPLIES THIS COLUMN ALREADY TRIANGULARIZED
C
      IF (A(L) .ne. 0.0D0) then
C
C        INTERCHANGE IF NECESSARY
C
         IF (L .EQ. K) GO TO 10
            T = A(L)
            A(L) = A(K)
            A(K) = T
   10    CONTINUE
C
C        COMPUTE MULTIPLIERS
C
         T = -1.0D0/A(K)
         CALL DSCAL(N-K,T,A(K+1),1)
      endif
c--------> out("pivot",k,l,t,a(k+1):(n-k))
      call outpiv(nt,k,l,n,t,a)
c--------> out("result",k,l,a:n)
      call outres(k,l,n,a)
      end
```

D Parallel Linpack 100 Code with Blocking

```
      SUBROUTINE DGEFA(A,LDA,N,IPVT,INFO)
      INTEGER LDA,N,IPVT(1),INFO
      parameter ( iwmax = 2**14 )
      DOUBLE PRECISION A(LDA,1), w(iwmax)
      DOUBLE PRECISION T
      INTEGER IDAMAX,J,K,KP1,L,NM1
C
C     GAUSSIAN ELIMINATION WITH PARTIAL PIVOTING
C
      data if/0/
      if ( if .eq. 0 ) then
         if = 1
         read(1,*) nt
```

```
         write(*,*)nt,' tasks.'
         close(1)
         if ( nt .eq. 0 ) call exit
      endif
      INFO = 0
      IF (N-1 .ge. 1) then
         call init(%val(n),%val(nt))
         do i = 1, nt
c --------> eval("evals",ccontrol(i,nt,n))
            call spawn(i,nt,n)
         enddo
         t1 = second()
         iwsize = min(iwmax,n*(n+nt-1)/nt)
         call grpcols(a,lda,n,iwsize,w,nt)
         write(*,*)second()-t1,' seconds to output columns'
         t1 = second()
         do k = 1, n + nt
c ------------------------> in("pivot",k,?ipvt(k),?t,?a(1,k):nn)
            if ( k .le. n ) ipvt(k) = inres(k,l,t,a(1,k))
c -------------------------> in("pivot",k-nt,?int,?double,?double *:)
            if ( k .gt. nt ) call inpiv ( k - nt )
         enddo
         write(*,*)second()-t1,' seconds to input results'
         do i = 1, nt
c --------> in("evals",?int)
            call cleanup
         enddo
      endif
      end
c---------------------------------------------------------------------
      subroutine grpcols(a,lda,n,iwsize,w,nt)
      double precision a(lda,*), w(n,*)
c
      nc = iwsize/n
      do it = 1, nt
         ic = 0
         iout = 1
         do j = it, n, nt
            ic = ic + 1
            do i = 1, n
               w(i,ic) = a(i,j)
            enddo
            if ( ic .eq. nc ) then
c -----------> out("col",it,iout,w:(n*ic))
```

```
               call outcol(it,n*ic,iout,w)
               iout = iout + ic
               ic = 0
            endif
         enddo
c ---------------------> out("col",it,iout,w:(n*ic))
         if ( ic .ne. 0 ) call outcol(it,n*ic,iout,w)
      enddo
      end
c-----------------------------------------------------------------
      function control(me,nt,n,aj,ak)
      double precision aj(n,*), ak(n), t
      mycol(k) = (k+nt-1)/nt
      kreal(k) = me + nt*(k-1)
C
C           ROW ELIMINATION WITH COLUMN INDEXING
C
      control = 0.0
c
c Read in my subset of the columns
c
      ncol = mycol(n)
      t1 = second()
      icol = 1
      do while ( kreal(icol) .le. n)
c -----> in("col",me,icol,?aj(1,icol):nn)
         call incol(me,icol,aj(1,icol),nn)
         icol = icol + nn/n
      enddo
      write(*,*)me,': ',second()-t1,' seconds to read columns'
      call update(me,nt,n,aj,ak)
      RETURN
      END
```

E C-wrapper code for Linpack

```
/* C wrapper code for plinpackd.f */
#
#include <stdio.h>
#include <time.h>
#
int ME = 0;
```

```
void init(n,nt)
   int n, nt;
{
   setlinebuf(stdout);
}

void spawn(me,nt,n)
   int *me, *nt, *n;
{
   int ccontrol();
   eval("evals",ccontrol(*me,*nt,*n));
}

int ccontrol(me,nt,n)
   int me, nt, n;
{
   int work_size;
   double *w, *ak;
   ME = me;
   setlinebuf(stdout);
   system("hostname");
   work_size = n*(n+nt-1)/nt;
   w = (double *) malloc(work_size*sizeof(double));
   ak = (double *) malloc(n*sizeof(double));
   control(&me,&nt,&n,w,ak);
}

void outcol(me,n,col,w)
   int *me, *n, *col;
   double *w;
{
   out("col",*me,*col,w:*n);
}

void incol(me,col,a,n)
   int *me, *col, *n;
   double *a;
{
   in("col",*me,*col,?a:*n);
}

void outpiv(nt,k,l,n,t,a)
   int *nt, *k, *l, *n;
   double *t, *a;
```

```
{
   out("pivot",*k,*l,*t,(a+(*k)):((*n)-(*k)));
}

void outres(k,l,n,a)
   int *k, *l, *n;
   double *a;
{
   out("result",*k,*l,a:*n);
}

int rdpiv(k,l,t,a)
   int *k, *l;
   double *t, *a;
{
   int nn;
   rd("pivot",*k,?*l,?*t,?(a+*k):nn);
   return(*l);
}

int inres(k,l,a)
   int *k, *l;
   double *a;
{
   int nn;
   in("result",*k,?*l,?a:nn);
   return(*l);
}

void inpiv(k)
   int *k;
{
   in("pivot",*k,?int,?double,?double *:);
}

void cleanup()
{
   in("evals",?int);
}
```