Linda in Adolescence

Robert Bjornson, Nicholas Carriero, David Gelernter and Jerry Leichter

Yale University
Department of Computer Science
New Haven, Connecticut

Introduction

Linda is a programming environment for building parallel applications and distributed systems on multi-computers [Gel85, CGL86]. It supports a high-level view of explicitly parallel programming, and its goal is to simplify the parallel programming task to the point where it is conceptually no harder than conventional, sequential programming.

Because of its high-level model, and the logically-shared tuple memory through which Linda processes communicate, implementing the language has long been regarded as a tough proposition -- particularly on multi-processors and local networks that provide no physically-shared memory among compute nodes.

We recently reported our first solution to the Linda implementation problem [CG85]. We described a Linda kernel that runs on AT&T Bell Labs' S/Net, a bus-connected multi-computer with no shared memory. We described tests that show Linda performing quite efficiently -- roughly on a level, in terms of network delay, with efficient message-passing kernels for bus-based networks, like the V kernel [CZ85]. Here we report on new work: (1) We have implemented new Linda kernels for the Intel iPSC hypercube and for a multiprogrammed VAX -- the latter includes a model for Linda implementations on shared-memory multiprocessors. (2) We have implemented a new C-Linda preprocessor that is designed to speed the runtime performance of all Linda kernels by analyzing communication patterns at compile time. These projects are all active and incomplete; our results and experience are preliminary. But substantial progress has been made in each case.

In the following we briefly describe Linda, and then outline the new projects.

Linda

Linda centers on an idiosyncratic memory model. Where a conventional memory's storage unit is the physical byte (or something comparable), Linda memory's storage unit is the logical tuple, where a tuple is just an ordered set of values. Where the elements of a conventional memory are accessed by address, elements in Linda memory haye no addresses; they are accessed by *logical name*, where a tuple's *name* is any selection of its values. Where a conventional memory is accessed via two operations, read and write, a Linda memory is accessed via three -- read, add and remove.

It is a consequence of the last characteristic that tuples in a Linda memory can't be altered *in situ*: to be changed, they must be physically removed, updated and then re-inserted. This in turn makes it possible for many processes to share access to a Linda memory simultaneously: using Linda we can build distributed data structures that, unlike conventional ones, may be manipulated by many processes in parallel. Furthermore, as a consequence of the first characteristic -- a Linda memory stores tuples, not bytes -- Linda's shared memory is coarse-grained enough to be supported efficiently *without* shared-memory hardware. But Linda is also a good match to shared-memory multi-computers, like the BBN Butterfly, IBM RP3, Encore Multimax and Sequent Balance machines. Its semantics are a high-level

version of the low-level semantics implicit in such architectures (in the sense that block-structured languages are high-level versions of stack machines).

Linda's shared memory is referred to as *tuple space* or TS. Messages in Linda are never exchanged between two processes directly; instead, a process with data to communicate adds it to tuple space, and a process that needs to receive data seeks it, likewise, in tuple space. There are three operations defined over TS -- `out()`, `in()` and `read()`. `out(`*t*`)` causes tuple *t* to be added to TS; the executing process continues immediately. Thus `out("foobar", 5)` generates tuple `("foobar", 5)` and adds it to TS. `in(`*s*`)` causes some tuple *t* that matches template *s* to be withdrawn from TS; the values of the actuals in *t* are assigned to the formals in *s*, and the executing process continues. If no matching *t* is available when `in(`*s*`)` executes, the executing process suspends until one is, then proceeds as before. If many matching *t*'s are available, one is chosen arbitrarily. Thus `in("foobar", formal 1)` may possibly remove tuple `("foobar", 5)` from TS, assigning 5 to formal parameter 1. `read(`*s*`)` is the same as `in(`*s*`)`, with actuals assigned to formals as before, except that the matched tuple remains in TS.

Any tuple element but the first may be a formal rather than a value, and any template element may be a value rather than a formal. When a template and tuple match, each value in the template is matched by a formal of the same type, or by an identical value, in the tuple; the same holds with respect to each value in the tuple. This rule resembles a symmetrical version of the select operation in relational databases.

We have argued at length that these communication primitives are highly flexible and powerful, and we have discussed matrix multiplication, LU decomposition and VLSI simulation experiments on the S/Net that make use of them [Gel85, CG86, CGL86].

The Cube experiment.

The Intel iPSC in the Computer Science Department at Yale consists of 128 Intel-80286-based processor nodes linked by dedicated Ethernet channels into a binary hypercube. The Cube kernel experiment has been useful and interesting, but the resulting system is more of an emulation, and a basis for code to be ported to other hypercube machines (an NCUBE machine will soon be installed), than an end in itself. This is so for two reasons. First, the iPSC lacks communication co-processors or front ends; each message packet interrupts each host along its route. We simulate an alternative by assigning half of the cube's nodes to act as hosts and the other half to act as dedicated communication processors. (New, soon-to-be-available hypercubes have real front-ends.) Reason two: the low-level send-message and receive-message primitives provided by Intel are painfully slow. Our Linda system relies on these primitives, and so it is painfully slow as well. Replacing the slow underlying primitives by fast ones should, obviously, make it faster.

Our present Cube kernel nonetheless makes a good basis for future Cube work. The system centers on a distributed hash table. Executing `out(`*t*`)` or `in(`*s*`)` causes tuple *t* or template *s* to be hashed (currently on the value of the first element) to some node in the communication network. An arriving tuple is checked against waiting templates; if there is a match, the tuple is shipped off to the node where the template was generated; otherwise it is kept for future use. Arriving templates are checked against stored tuples and buffered in case of no match in exactly the same way.

We've experimented with Cube Linda using a simple matrix multiplication program similar to what is described in [CG85]. Rows of the first multiplicand and columns of the second are stored in tuples; a group of identical worker processes repeatedly `in`'s a "next task" tuple, performs the computation requested, and loops until there are no more tasks. Figure 1 shows results for a banded version that computes several rows of the product per task: ten workers plus one control process finish approxiately five times faster than a single node running a sequential C program.

Is it possible to build a Cube Linda as efficient as S/Net Linda? We still can't say. Porting the Cube kernel to the lower-overhead NCUBE will give us a better idea of its potential; faster communication will

ultimately come to the Intel machine as well.

The VAX kernel

The VAX implementation assumes a two-level model. At one level, it assumes multiple processors sharing a common memory. There is little real difference between this and a uni-processor running multiple user processes; it just requires more care in getting the synchronization right. Modern operating systems provide synchronization services, but they are not intended for implementing systems like Linda, which may require synchronization every couple of hundred instructions. The underlying hardware also provides synchronization operations; there may be two orders of magnitude faster, but are difficult to coordinate with the OS-level operations -- when using a shared machine, a hardware spin lock wastes resources perhaps useful to other processes. We avoid this problem by using hardware locking mechanisms for the common cases, and having the code fall back to OS mechanisms only when necessary. For example, access to individual tuples is synchronized using hardware mechanisms and spin locks. It is assumed that no process will hold on to a tuple lock for very long. On the other hand, when tuple space must undergo wholesale reorganization, an OS-level lock is taken out.

At the second level, a number of these multi-processors are assumed to be connected by a relatively slow, somewhat unreliable link, such as an Ethernet. This two level model corresponds to an increasingly common computing environment: A LAN with workstations and "compute servers". The VAX implementation is designed to allow sharing of both the "compute server" and the LAN - and, for that matter, the "workstations" - with other user processes, running Linda or anything else. The hardest problem we've had to deal with is the unreliability of Ethernet broadcasts. The original S/Net implementation used what we label a positive broadcast scheme: The `out()` operation sends the data in the tuple to all nodes. The semantics of Linda, in which an `out()` completes immediately, make it very difficult to deal with loss of one of these `out()` packets. In the VAX implementation, we use a negative broadcast scheme: The data from out()'s is stored locally, and `in()`'s (or `read()`'s) broadcast a request to all nodes. Since Linda semantics specify that `in()`'s and `read()`'s wait for a matching tuple, loss of such a request can be handled by having the kernel, which remains in control until it is able to satisfy the `in()` or `read()`, try again later.

Each of the two levels of the implementation is of interest in its own right. The shared memory level provides a model for implementations on the many shared memory machines that hardware architects seem to be designing these days. The LAN level will allow us to experiment with the use of Linda for systems programs, such a mailers.

The new pre-processor.

Linda derives much of its power from the dynamic, flexible character of its tuple-template matching algorithm. `out` may build a tuple out of any combination of values and typed formals, and `in` or `read` may select one based on any combination of values and typed formals. Such flexibility is expensive at runtime, and we'd like to make the kernel faster without weakening the language.

There are several ways to do so. One possibility is to support tuple-template matching in hardware. A Linda chip set is now in design (as a collaborative project, mainly involving Venkatesh Krishnaswamy of our group and Sid Ahuja of Bell Labs). An allied effort involves compile-time analysis. A compiler that examines all Linda operations in a source program can establish several things that are of use at runtime. When it can be established that a given `in` or `read` can only be satisfied by tuples contributed by a given set of `outs`, we can restrict runtime matching to the eligible tuples only. It can sometimes be further established that certain fields in a template will *always* match corresponding fields in a tuple; at match-time, we can ignore these fields altogether.

The current Linda pre-processor works for C-based Linda (i.e., C with Linda primitives added), and is based on the first pass of the portable C compiler. Analysis works in two stages -- first individual modules are analyzed, then the per-module results are unified and linked -- so that changes to one module

don't require re-compilation of the whole program.

In coming weeks (as of early April) we will integrate the pre-processor and the S/Net kernel, and see what we get. Note that the pre-processor has a special role to play in conjunction with the Cube kernel: when we can establish that tuples of a given type are of interest only to one process, we can customize our hash function in such a way that tuples of that type are hashed directly to the node on which the process that needs them is loaded.

References.

[CG85] N. Carriero and D. Gelernter, "The S/Net's Linda Kernel," in *Proc. ACM. Symp. Operating System Principles*, (Dec. 1985) and *ACM Trans. Comp. Sys.* (May 1986) (to appear).

[CGL86] N. Carriero, D. Gelernter and J. Leichter, "Distributed data structures in Linda," *Proc. ACM Symp. Principles of Prog. Languages*, Jan. 1986.

[CZ85] D.R. Cheriton and W. Zwaenepoel, "Distributed process groups in the V kernel," *ACM Trans. Comp. Sys.* 3,2(1985):77-107.

[Gel85] D. Gelernter, "Generative communication in Linda," *ACM Trans. Prog. Lang. Sys.* 1(1985):80-112.

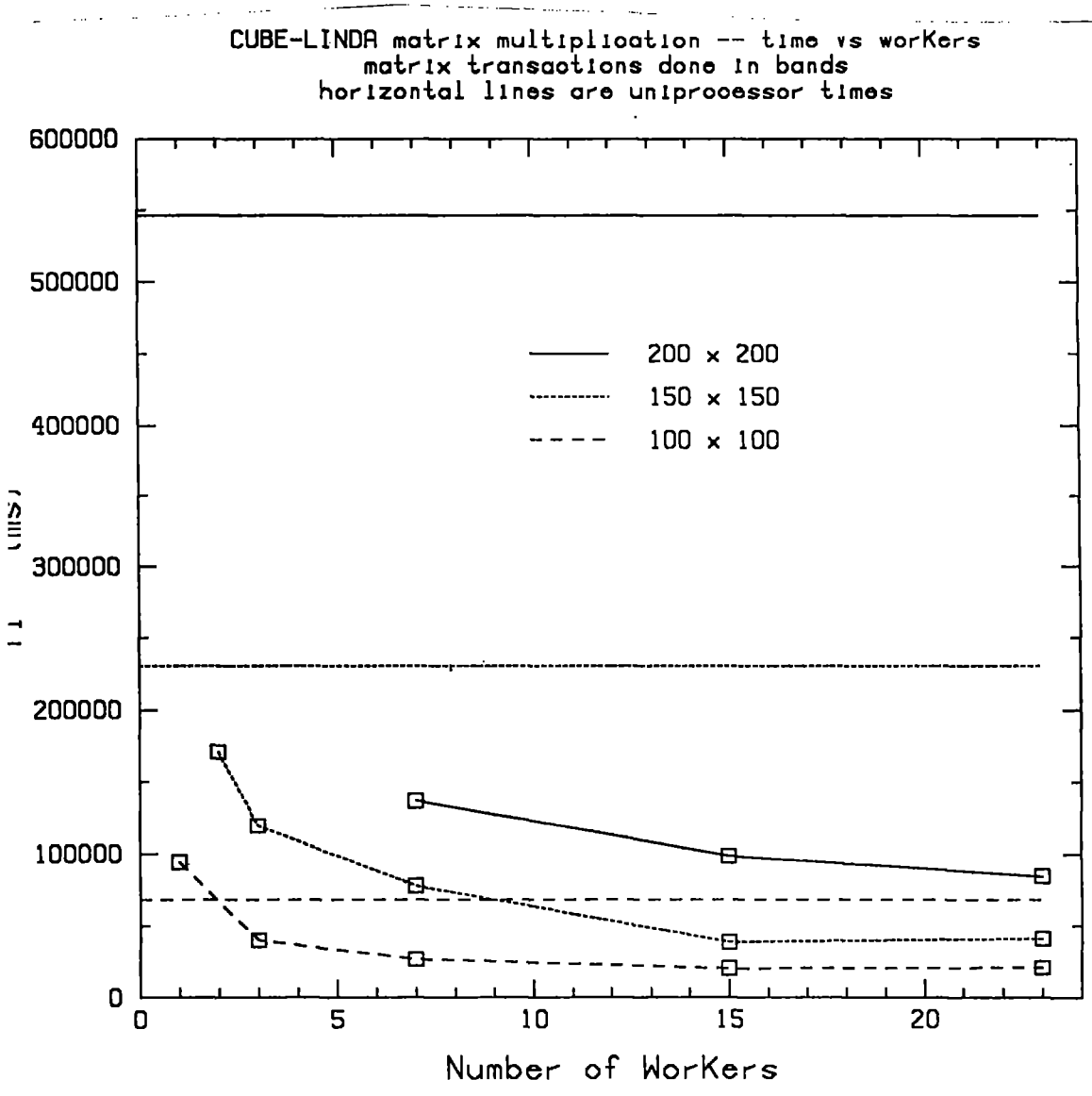

figure 1.