The S/Net's Linda Kernel

Nicholas Carriero and David Gelernter
Yale University
Department of Computer Science
New Haven, Connecticut

extended abstract

Linda consists of a few simple operators that, when injected into a host language *h*, turn *h* into a parallel programming language. (Most of our programming experiments so far have been conducted in C-Linda, but we have recently implemented a Fortran-Linda preprocessor as well.) The S/Net is a multi-computer that can also function as the backbone of a local area net; each S/Net is a collection of not more than sixty-four memory-disjoint computer nodes communicating over a fast, word-parallel broadcast bus. We have implemented a Linda-supporting communication kernel on an S/Net at AT&T Bell Labs (where the machine was designed and built), and this implementation is interesting, we argue, for two reasons. It demonstrates, first, that Linda's powerful and flexible communication primitives can be made to run well; the language's shared-memory-like semantics can in fact be supported in the *absence* of physically shared memory. Second, although Linda and the S/Net are particularly well-matched, the simplicity of the language, of the implementation's design and of the S/Net's logical structure suggest that Linda implementations might readily be constructed on similar architectures elsewhere. A Linda kernel similar to the S/Net's has in fact recently been brought up, in preliminary form, on an Ethernet-based network of Micro-Vax workstations.

Processes in Linda communicate through a globally-shared collection of ordered tuples called tuple space or TS. The operators that Linda provides add tuples to this shared collection, remove tuples and read tuples. `out`(*t*) causes tuple *t* to be added to TS; the executing process continues immediately. `in`(*s*) causes some tuple *t* that matches template *s* to be withdrawn from TS; the values of the actuals in *t* are assigned to the formals in *s*, and the executing process continues.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. `read`(*s*) is the same as `in`(*s*), with actuals assigned to formals as before, except that the matched tuple remains in TS.

Our S/Net implementation of these operations buys speed at the expense of communication bandwidth and local memory; the reasonableness of this trade-off was our starting point. In the kernel scheme we implemented (many others are possible), executing `out`(*t*) causes tuple *t* to be broadcast to every node in the network; thus every node stores a complete copy of TS. Executing `in`(*s*) triggers a local search for a matching *t*. If one is found, the local kernel asks *t*'s origin node for permission to delete it; if permission is granted, *t* is returned to the process that executed `in()`. (Permission is denied only when some other process is attempting to delete *t* more-or-less simultaneously, and *it* has been told to go ahead.) If the local search triggered by `in`(*s*) turns up no matching tuple, all newly-arriving tuples are checked until a match occurs, at which point the matched tuple is deleted and returned as before. `read()` works in the same way as `in()`, except that no tuple-deletion need be attempted; as soon as a matching tuple is found, it is returned immediately to the reading process.

We have studied the kernel's performance on a small (8 node) S/Net by measuring running times for processes doing communication only, and by experimenting with a simple matrix multiplication program in the promising but relatively unfamiliar distributed-data-structure style. Communication delays in our system are comparable in rough terms (given its idiosyncratic character, and our hardware) to delays in efficient message-passing kernels for bus-based networks like Birrel and Nelson's RPC kernel and Cheriton and Zwaenpoel's V kernel.

Linda offers parallel programmers a new way of looking at network communication systems. Standard communication protocols require that information be handed around from process to process; no process can unburden itself of new data without first determining where the data should go, and then handing it along explicitly. Linda processes, on the other hand, are anonymous drones sharing access to one data pool. Shared memory has long been regarded as the most flexible and powerful way of sharing information among parallel processes -- but a naive shared memory is hard to implement without hardware support, and requires the addition of synchronization protocols if it is to be safely accessed in parallel. In Linda, however, the shared memory's cell-size is the logical tuple, not the physical byte, and so it is coarse-grained enough to be supported efficiently without special hardware. And because, in Linda's shared memory, data may not be altered *in situ* -- it is accessible via read, remove and add instead of the standard read and write -- it may safely be shared by any number of parallel processes. Although much work needs to be done and our work with the Linda kernel is still at an early stage, we have taken (we feel) a significant first step towards demonstrating the power and practicality of Linda as a parallel programming model.

This work was performed primarily at AT&T Bell Labs, Holmdel; Sid Ahuja, Erik DeBenedictis, Robert Gaglianello, Howard Katseff and Thomas London, all of Bell Labs, were our research collaboraters. This material is based upon work supported by the National Science Foundation under Grant No. MCS-8303905.