

The Distributed Dataflow Language vLucid

W. H. Mitchell
Microsoft Corp.
Microsoft Way
Redmond, WA 98073
billmit@microsoft.com

A. A. Faustini
Computer Science Department
Arizona State University
Tempe, Arizona 85287
(602) 965-3983
faustini@lu.eas.asu.edu

1 Introduction

Visual Lucid (vLucid) is a high-level distributed visual dataflow programming language which enables the construction and evaluation of programs that are *implicitly* and/or *explicitly* distributed. Individual program modules at any granularity level may be shipped off for remote evaluation to other Macintosh® computers on an Appletalk® network. Target processors may either be prespecified at program construction time, or automatically selected from the available target processor pool.

Based on the textual, demand-driven dataflow language *Lucid* (Wadge and Ashcroft 1985), vLucid affords programmers greater ease of program construction and aids in program and data visualization. Additionally, it allows them to realize the potential parallelisms afforded by the dataflow paradigm on commonplace microcomputer networks. Although most dataflow research has concentrated on producing hardware and software that automatically takes advantage of fine-grained parallelism, the distributed nature of vLucid provides more of an extensible distributed programming environment in which both coarse-grained and fine-grained operators may peacefully coexist.

1.1 Overview of Lucid

Lucid is a higher-order (Wadge 1991) functional language whose syntax is based on Landin's nested *where-clause* structure (Landin 1966). Lucid's underlying data algebra $Lu(A)$ is based on a simpler algebra A whose operators have all been extended to operate in a pointwise manner, which then has been enlarged to include additional *stream operators* such as *first*, *next* and *fb*. Interested readers are directed to (Mitchell 1988) and (Orgun and Wadge 1989) for accounts of Horn-clause logic based languages which make use of similar stream operators.

Lucid's operators allow programs to operate conveniently on streams of data values, acting much as a connected set of "black box" filters. A program that transforms a continuous stream of characters into a stream of object code can thus be visualized as depicted in figure 1.



Figure 1 - Compilation in Lucid using “black box” filters.

More interesting program behavior, such as *feedback*, may be achieved through the use of stream operators. For example, we may write a simple program that generates the stream of all natural numbers as:

```

nats where
  nats = 1 fby nats + 1;
end

```

The `fby` operator merely computes as its result a stream composed of the first element of the stream applied as its left-hand-side argument, “followed by” the entire stream applied as its right-hand-side argument. It is instructive to consider this program in *dataflow graph* form, as depicted in figure 2. In this form we may readily observe the feedback nature of the program¹, where the operator `fby` sequences the stream produced as output. The stream’s first value is 1, which is followed by this value plus 1, which is followed by that value plus 1, and so on, ad infinitum.

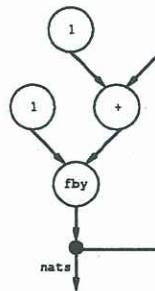


Figure 2 - Natural number generation in Lucid.

Each data element or *daton* in a Lucid stream may itself be a stream. Each such stream corresponds to a dimension in *intensional space* (Thomason 1974). Each dimension has associated with it various *intensional* (or *stream*) operators which potentially permute the stream in that dimension. By convention, the primary dimension is referred to as *time*, and each of the other dimensions as *space*, with various indices.

¹ The filled circle node that receives input from `fby` represents the `split` operator.

<i>Dimension</i>	<i>Operators</i>			
TIME	<code>first</code>	<code>next</code>	<code>fbv</code>	...
SPACE0	<code>init</code>	<code>rest</code>	<code>sby</code>	...
SPACE1	<code>init1</code>	<code>rest1</code>	<code>sby1</code>	...
:	:	:	:	:

Figure 3 - Some of Lucid's *intensional* operators.

Interesting programs that take advantage of the multiple-dimensionality of Lucid may be quite compactly constructed using the usual textual Lucid representation. Consider, for example, a program to compute prime numbers using the *Sieve of Eratosthenes* approach:

```

first primes where
  primes = nums sby primes wvr primes mod first primes ne 0;
  nums = 2 fby nums + 1;
end

```

As may be expected, the *dataflow graph* form of such a program is rather easier to grasp, providing vLucid with one of its major advantages over textual dataflow languages. Additionally, however, the graphical nature of vLucid also provides the framework needed to very easily construct various types of distributed programs, as we will discuss in section 4.

1.2 Organization of This Paper

The rest of this paper consists of three parts: section 2 describes the sources of potential parallelism available to us in a dataflow environment like vLucid, and provides motivation for the implementation. Section 3 is devoted to an overview of the vLucid language and programming environment. In section 4 we discuss the actual distributed programming mechanisms of vlucid and their relative merits. Section 5 concludes with a discussion of related work and a summary of the noteworthy features of this system.

2 Dataflow Parallelism and Distribution

Dataflow languages contain several varieties of implicit, easily detectable potential parallelism, which we refer to respectively as *temporal*, *spatial* and *structural concurrency*, after (Davis and Keller 1982). These parallelisms are easiest to illustrate when directly considering a particular dataflow graph, so we now introduce the graph of figure 4 which represents the expression $x^2 - 2x + 3$.

The most obvious concurrency exhibited in this figure is *pipeline* or *temporal concurrency*: Whole streams of datons move along the arcs, although generally nodes only operate on a single daton from each input at a time.

A second form of parallelism we may observe in this illustration is *spatial concurrency*. Note that the execution of operators that do not depend on each other for input may be scheduled to occur simultaneously. Thus, for instance, both multiplication operations may be performed in parallel.

To understand the final variety of parallelism, *structural concurrency*, we must envision the datons flowing along arcs in the above graph to be complex data objects such as vectors or matrices instead of simple numbers. In this situation, we can also envision operators in the dataflow graph, such as the node labeled +, as functions which may perform parallel operations on complex data types (e.g. vector addition).

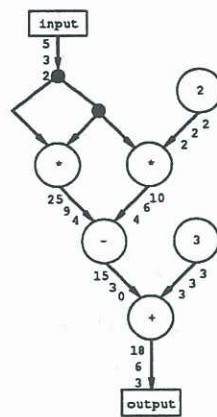


Figure 4 - Dataflow graph for the expression $x^2 - 2x + 3$.

2.1 Modes of Distribution

The three different modes of parallelism just discussed essentially serve to inspire a corresponding set of modes of potential distribution. First, *temporal distribution* involves the creation and distribution of n copies of an operator Θ , along with a complete set of datons for each operator, for steps $0 \dots n$ of the dataflow computation. We may entertain a number of options in achieving this parallelism, depending on how we chose to implement concurrency and distribution, as depicted in figure 5.

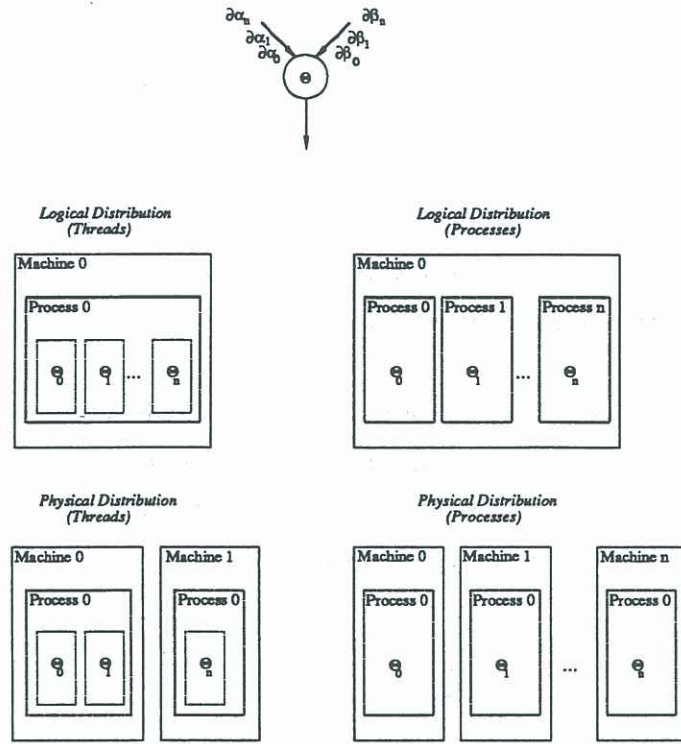


Figure 5 - Temporal distribution techniques.

Under *process-level concurrency* a new process is created, either on the same machine (logical) or on another networked machine (physical) for every operator instance Θ_i required to process a set of data elements from its input streams $\partial\alpha_i, \partial\beta_i, \dots, \partial\gamma_i$. This is fairly straightforward to implement and may easily be achieved through the use of RPC (Nelson 1981) style techniques like those used in GLU (Jagannathan and Faustini 1990). Unfortunately, this technique pays a high price in terms of the overhead necessary for process creation and disposal and interprocess communication.

A more difficult to implement but more efficient method employs *thread-level concurrency*. Under this model a new thread (that is, subportion of a process with its own stack but a shared heap) is spawned corresponding to each operator instance required, on the same machine and inside the same process (logical) or on another machine (physical).

It should be noted that the language Lucid, as pointed out in (Ashcroft, Faustini and Jagannathan 1991) stands to benefit much more from this sort of pipelined distribution than other dataflow languages due to its intensional nature. Since the key data structure in Lucid is the n-dimensional multi-stream, a relatively simple (textually or

graphically speaking) Lucid program may specify a vast amount of tagged daton movement and computation, much of which may be performed in parallel using this technique.

Spatial distribution, whereby operators which are free from interdependencies may be computed concurrently, is also a viable dataflow concurrency mechanism. Figure 6 illustrates spatial distribution and the various means of achieving it. Again we have choices based on the use of either processes or threads and based on whether to simulate or actually physically distribute computation.

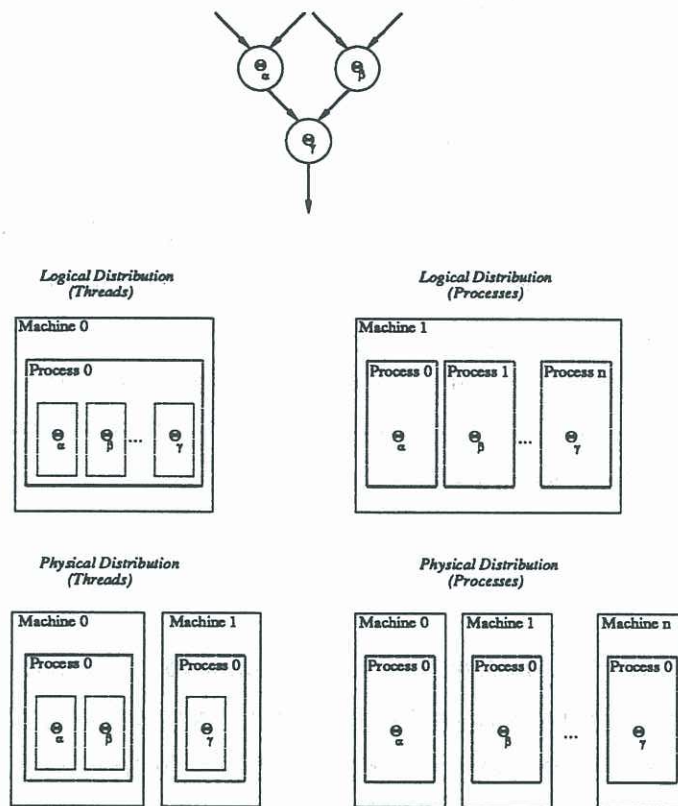


Figure 6 - Spatial distribution techniques.

The final mode of distribution available is *structural distribution*. This mode requires fairly complex operators (eg. matrix operations) which operate quite possibly on large and abstract data types. Under this mode of distribution such operators would distribute sub-portions of the decomposable complex operator's computations to sets of finer-grained operators on different machines and/or in different tasks and processes.

Two observations need to be made in regard to the applicability of this mode. First, such structured distribution may be simulated through the appropriate use of UDFs and spatial distribution. For instance, a matrix addition

component may easily be constructed from multiple scalar addition operators and appropriate *selection* and *construction* functions².

Secondly, vLucid is completely devoid of any sort of ADT facilities and Lucid in general encourages more the use of intensionality and multistreams of primitive datons (i.e. in the representation of a matrix) rather than the use of large, complex datons (although see (Freeman-Benson 1991) for a language proposal counter to this).

For these reasons we dismiss structural distribution from further consideration in the context of distributed visual dataflow.

3 The vLucid Language

vLucid supports the direct visual programming of Lucid programs in dataflow graph form. The prototype implementation that we have constructed runs on the Macintosh® family of microcomputers and supports a subset of the pLucid (Wadge and Ashcroft 1985, Appendix) implementation's datatypes and operators. Operators are represented by corresponding graphical icons which may be mouse-manipulated in the same manner as objects in typical paint or draw programs. Arcs may be rubber-band-drawn between operators to connect them, and graphs thus constructed may be encapsulated to form user-defined functions (UDFs).

3.1 Basic Types and Operators

The two data types supported in vLucid are *real* and *boolean*. The operators of vLucid can be classified as falling into one of four categories: *numeric*, *boolean*, *conditional*, and *stream operators*, as illustrated in figures 7-10. Each of the numeric operators accept as input, and produce as output, 32-bit floating point numbers. Boolean operators, similarly, operate on and produce boolean values. Conditional and stream operators are polymorphic and operate on streams of values of arbitrary type³. Figure 10 shows other special operators required, and several sample UDFs. The basic vLucid *input* operator causes a dialog box to be popped up, requesting a boolean or numeric value. Conversely, the basic *output* operator creates and displays a window showing the output value stream.

² A *select* (*m*, *i*, *j*) function returns the value of the (*i*, *j*)*th* element of the matrix *m* and a *construct* (*n*, *i*, *j*, *v*) function returns a new matrix *m* exactly the same as the old matrix *n* except for the value of the (*i*, *j*)*th* component, which is now *v*.

³ Actually, the *if-then-else* operator accepts only booleans as its leftmost input, and the *@t* operator rounds its rightmost input to the nearest integer value.

In addition to the I/O operators, the *Other Sheet* window of figure 10 shows several UDFs of various arity: *foo*, *bar*, *bleen*, *zug* and *ferd*. Methods for their creation and use will be treated shortly. Also, several constant-generating nodes, namely *true* and *3.14* are shown. Obviously constant-generators are not provided for all of the infinite number of constants possible; instead, a single constant-generator that may be configured by the user is provided. A double-click on the constant generator's icon pops up its configuration dialog.

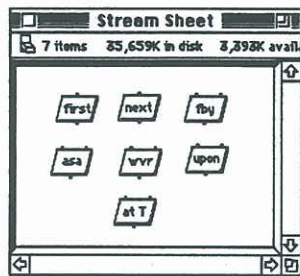


Figure 7 - vLucid stream operators.

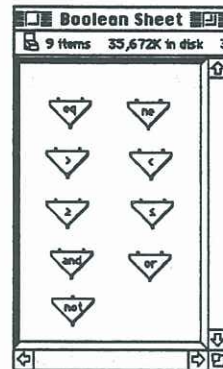


Figure 8 - vLucid boolean operators.

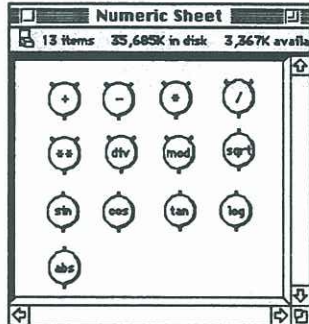


Figure 9 - vLucid numeric operators.

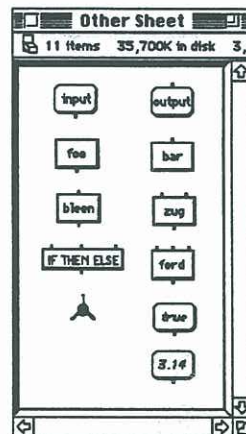


Figure 10 - Misc. other vLucid operators.

3.2 vLucid Program Construction

Programs in vLucid are constructed within *program sheet* windows, as shown in figures 11–14, which correspond to user defined functions (UDFs). Program sheets exist as separate data files at the operating system level, although heirarchical components are all stored within the same file. Sheets on disk may be opened, and new sheets may be created through suitable menu selections.

Groups of operators from a program sheet may be selected (via *marquees* or *shift-clicks*) and the **Encapsulate** menu item from the **Options** menu subsequently selected to create a new UDF and corresponding program sheet containing these operators. The icon corresponding to the new UDF logically appears in place of the operators that were encapsulated, and the function arity of the new operator is determined automatically from the number of inputs and outputs of the operator group selected for encapsulation.

vLucid users are encouraged to employ nullary program sheets, that is, UDFs without any inputs/outputs, as the basis for *operator toolkits* or *libraries*, much as we have shown in figures 7–10. This affords the user an operator palette-like selection mechanism.

3.3 An Example Program: $n!$

The program illustrated in figures 11–14 computes the factorials of the stream of input numbers entered, and displays this stream in an output dialog window. This program has been constructed in a top-down manner: initially, as shown in figure 11, the `nFact` component is merely a function shell. Then, in figure 12, it is filled with components provided the desired functionality, and wired appropriately. The specification of `fact` and `nats` proceeds similarly.

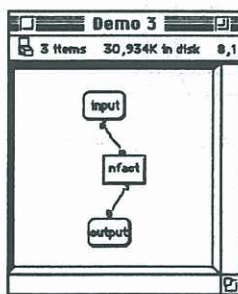


Figure 11 - Factorial program.

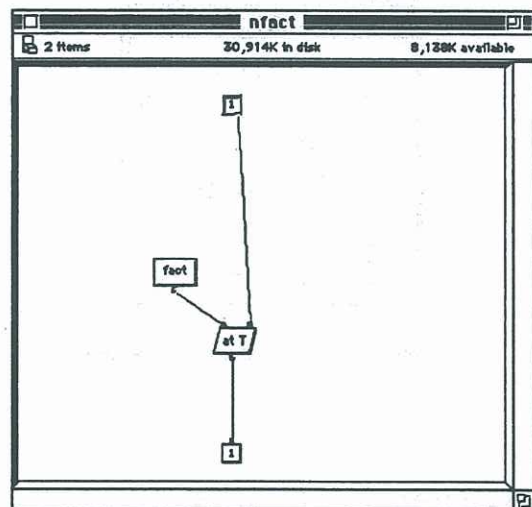


Figure 12 - `nFact` module which generates $n!$.

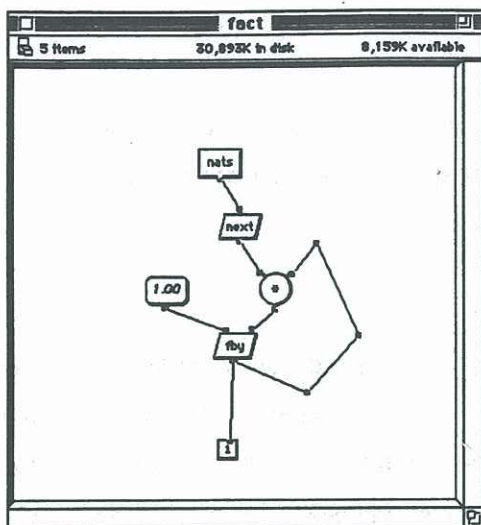


Figure 13 - Fact module which generates a factorial stream.

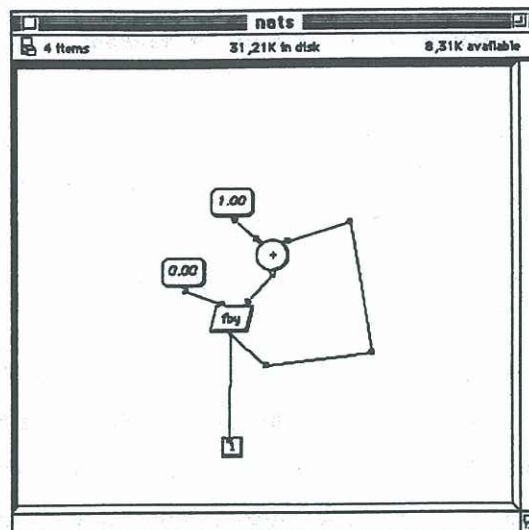


Figure 14 - Nats module which generates a stream 0...n.

3.4 vLucid Environment

Intuitively, one might expect a **Run** menu item to be available to trigger the evaluation of vLucid programs. After careful consideration, however, it should be apparent that the environment would need a parameterized **Run**, since multiple, non-nested program sheets may be opened at any one time, and each logically corresponds to what textually would be considered a completely separate dataflow program. Instead, we would like to maintain selectivity over which of these programs we evaluate.

Each vLucid program sheet contains a *switch*, like a common light switch, in its upper left-hand corner. This switch may be toggled at any time and controls the execution of the program within its enclosing program sheet. The switch sends out a *meta-demand* to all of the output operators in the program sheet. An attribute of each of these output operators controls their individual demand patterns once they have received such a meta-demand. The driving of the evaluator, hence, is completely controlled by individual output operators, which may be configured via pop-up attribute dialogs invoked by double-clicking. Program execution may be terminated at any time by the user through a simple mouse-press that turns the switch OFF. Even more interestingly, evaluation may be triggered in multiple selected sheets, permitting pseudo-simultaneous execution of different programs.

4 Distributed Programming in vLucid

In section 2 we discussed some of the attributes of dataflow programs which make them particularly amenable to distributed execution. Now, we turn our attention to vLucid distributed programming techniques specifically.

4.1 Explicit Distributed Programming

A particularly simple approach to distributed computation is that which we refer to as the *Blind Producer/Consumer (BPC)* model. We leverage this approach on the fact that all I/O in the type of GUI-based OS environment we rely upon is made available through OS event queues. *Inter-application communication (IAC)* events also travel through these queues, giving us the infrastructure needed to design *explicitly distributed* programs: those in which program operator distribution is a part of the specification of the program itself.

By providing primitives that are capable of both “pushing” onto and “pulling” from application event queues and the OS event queue, we provide a mechanism by which programs on different, networked machines may communicate, and cooperate in achieving some desired computation.

Consider the case in which we would like to approximate the value of e^x for some value of x using Taylor Series expansion. The simple formula

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^{n-1}}{(n-1)!}$$

yields approximations to the value of e^x with accuracy that increases with the number of terms in the expansion. Assuming that some other computing node on the network we are connected to is capable of producing factorials at a much faster rate than we can on our machine, and further, that this machine’s user has agreed to produce factorials for us, a producer–consumer pipeline may then be set up, as illustrated in the screen snapshots in figures 15 and 16.

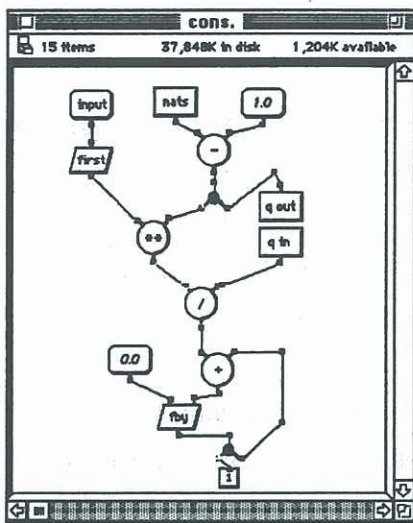


Figure 15 - Factorial *consumer* vLucid program.

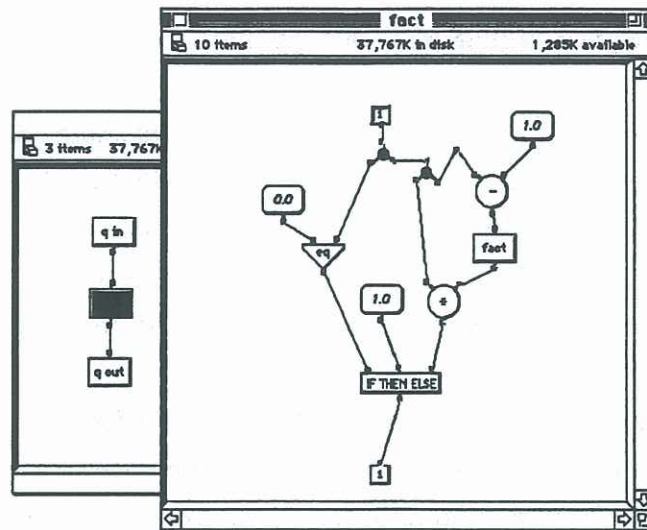


Figure 16 - Factorial *producer* vLucid program.

Several important points must be made about this mode of distributed programming. First, it is apparent that producer—consumer synchronization is, at the very least, sorely lacking. In fact, it is only due to the fairly generous system and program I/O queues⁴ that take up some of the communications slack, coupled our use of a tagged dataflow scheme which operates correctly in the presence of unordered data-stream tokens, that this mode works at all. Even so, producers and consumers must get together to time their production and consumption, at least approximately, since IAC events will eventually be dropped from the queues when they fill up.

This brings us to another problem inherent in the BPC model: lack of fault tolerance and any sort of deadlock detection or prevention capabilities. If even one event corresponding to a certain required tagged daton value is dropped from an input queue where it is needed, or for some reason it never even arrives there, the consumer will continue to wait on this value forever. Worse, deadlock (or a *deadly embrace*) can occur if communicating programs act as both producers and consumers of complementary values which are needed in the production/consumption of some later stream values.

Despite these problems, this primitive model affords us several interesting experimentation opportunities, including the chance to investigate primitive *coprogramming*. Coprogramming involves the joint dynamic development of software through *cooperative programming* efforts, using networked computers. As we have seen, use of the BPC model almost enforces coprogramming practices, since users are responsible for providing all producer—consumer synchronization.

⁴ At least on the Macintosh under System 7.

4.2 Implicit Distribution through Program Annotation

The distribution of program fragments in distributed systems based on RPC and RPC-like mechanisms (Nelson 1981) such as in GLU (Jagannathan and Faustini 1990) and TDFL (Suhler et al. 1990) is similar to the technique used by the blind producer/consumer model. Procedures, assumed to be pre-existent on remote machines, are called via a simulated “normal” procedure call interface. The results of these remote procedures are computed, then sent back to the originator, but the code for the procedure remains on the same machine, ready to be run again.

An alternative to this approach is to base the distribution of functions on *remote evaluation (REV)* (Stamos and Gifford 1990). vLucid provides a distribution method based on such a technique, which we refer to as the *Program Server (PS)* model of distribution. To achieve this behavior, the vLucid environment is divided into two components: *native space* and *foreign space*, as shown in figure 17. Native space supports the visual vLucid environment running on the host machine and handles execution of *native programs* (i.e. those not distributed to another machine). Foreign space is where functional program components that have been distributed to this particular host machine are executed. Their execution is transparent to the user of the host vLucid environment, who has no access to any visual representation of them whatsoever.

Under the PS model vLucid users may *statically* (that is, at program construction time) distribute the graphical dataflow programs they construct manually via simple, direct-manipulation-based user-directed *mapping* (or *partitioning*). To map elements of a program to different processors, a user merely selects the icons corresponding to the functions targeted for distributed execution, then chooses the **Distribute** menu item. Doing so will cause a dialog-box to pop-up, requesting selection of a target processing node. The complete process is illustrated through the image sequence of figure 18.

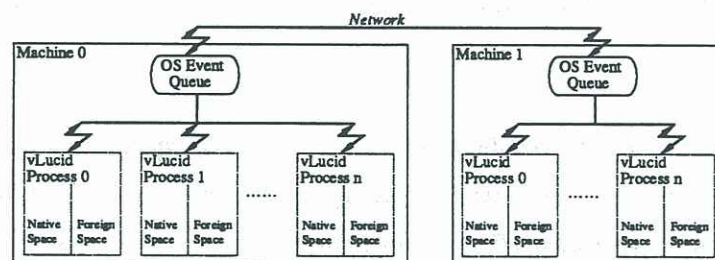


Figure 17 - vLucid PS-model distribution mechanism.

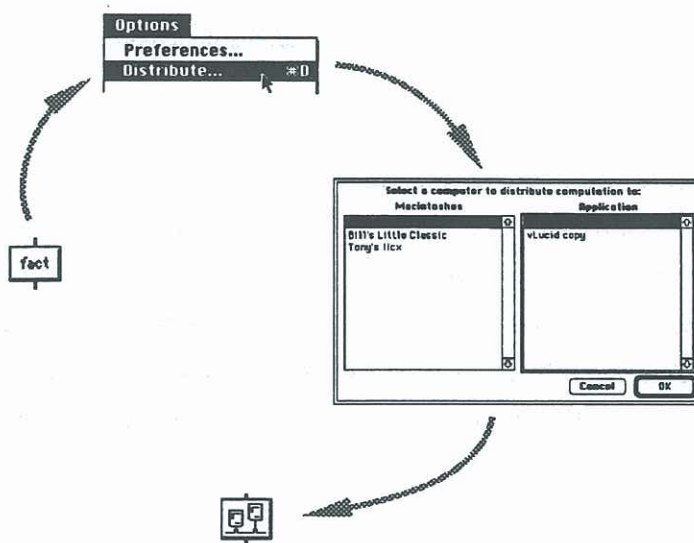


Figure 18 - Schematic view of user-directed distribution partitioning.

While in terms of simplicity, the manual distribution facility just described is an advance over other manual mapping schemes such as the textual annotations of Hudak (1986), automatic optimal mapping based on load balancing considerations is typically considered the “holy grail”. We do not attempt to provide such a mechanism in vLucid, since one of our goals is instead to permit the user more control over the distribution.

4.3 Implicit and Dynamic Distribution

The implicit distribution just discussed is a mechanism by which vLucid programmers may take advantage of spatial concurrency, as related in section 2. While this mechanism has the advantage of allowing the user to directly control computation resources used by their programs, an alternate mechanism similar to that provided in GLU, which takes advantage of temporal concurrency, is also provided.

Under this mechanism a process similar to that shown in figure 18, for distributing program modules to explicitly specified execution targets is used, but instead of selecting a single processing node, the user selects a subset of n nodes which may then all be used (in a round-robin fashion) to process demands for different data elements at different points in the computation stream.

5. Summary

The key contribution of vLucid is its simplification of the distributed programming task. While some of the credit for this achievement belongs to the dataflow paradigm in general for smoothing over traditionally sticky issues like

process communication and synchronization, the graphical, direct-manipulation nature of the vLucid system also deserves moderate credit.

Several other research systems provide visual dataflow-graph programming including Fabrik (Ingalls et al. 1988) and early systems developed at the University of Utah (Keller and Yen 1981) (Davis and Keller 1982), but distributed visual languages are still a relative novelty. While much previous work has been done in visual programming, distributed programming, and dataflow, the only system we know of combining these three approaches is TDFL (Suhler et al. 1990). vLucid differs from TDFL in that it relies on an REV-like mechanism instead of RPC to allow a more dynamic distribution process. Also, vLucid is based on a *demand-driven* or *lazy* semantics, as opposed to the *data-driven* semantics of TDFL.

References

- Ashcroft, E. A., Faustini, A. A., and Jagannathan, R. 1991. An intensional parallel processing language for applications programming. In *Parallel functional languages and compilers*, ed. B. K. Szymanski. Reading, MA: Addison-Wesley.
- Davis, A. L. and Keller, R. M. 1982. Data flow program graphs. *IEEE Computer* 26, 2 (February): 26-41.
- Freeman-Benson, B. N. 1991. Lobjcid: Objects in Lucid. In *ISLIP '91 Proceedings* (April): 80-87.
- Gaudiot, J. and Bic, L. 1991. *Advanced topics in data-flow computing*. Englewood Cliffs, NJ: Prentice Hall.
- Hudak, P. 1986. Para-functional programming. *IEEE Computer* 19, 8 (August): 60-70.
- Ingalls, D., Wallace, S., Chow, Y. Y., Ludolph, F. and Doyle, K. 1988. Fabrik, a visual programming environment. In *OOPSLA 88 Conference Proc.* San Diego CA (September): 176-190.
- Jagannathan, R. and Faustini, A. A. 1991. GLU: A system for scalable and resilient large-grain parallel processing. Draft Tech. Report, SRI Computer Science Laboratory (May): Menlo Park, CA.
- Keller, R. M. and Yen, W. J. 1981. A graphical approach to software development using function graphs. *Digest of Papers, Compcon Spring '81*: 156-161.
- Landin, P. J. 1966. The next 700 programming languages. *Communications of the ACM* 9, 3 (March): 157-166.
- Mitchell, W. H. 1988. *Intensional horn clause logic as a programming language - It's use and implementation*. M.S. Thesis, Arizona State University, Tempe, AZ (December).
- Nelson, B. J. 1981. Remote procedure call. Tech. Report CMU-CS-81-119, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh PA (May).
- Orgun, M. A. and Wadge, W. W. 1989. Chronolog: A temporal logic programming language and its formal semantics. Unpublished manuscript (January).
- Stamos, J. W. and Gifford, D. K. 1990. Remote evaluation. *ACM Transactions on Programming Systems and Languages* 12, 4 (October): 537-565.
- Suhler, P. A., Biswas, J., Korner, K. M. and Browne, J. C. 1990. TDFL: A task-level dataflow language. *Journal of Parallel and Distributed Computing* 9 (September): 103-107.
- Thomason, R. ed. 1974. Selected papers of R. Montague. In *Formal philosophy*. New Haven, Conn.: Yale University Press.
- Wadge, W. W. 1991. Higher order Lucid. In *ISLIP '91 Proceedings* (April): 62-69.

Wadge, W. W. and Ashcroft, E. A. 1985 *Lucid, the dataflow programming language*. London, England: Academic Press.