Canister Communication in Parallel Programs*

Duane A. Bailey
Janice E. Cuny

COINS Technical Report 88-42
October 1988

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

*The Parallel Programming Environments Project at the University of Massachusetts is supported by the Office of Naval Research, contract N000014-84-K-0647.

Abstract

Global communication patterns are fundamental to our understanding of massively parallel, closely-coupled computation: streams of related data are pipelined along sequences of channels, values are broadcast to (and aggregated from) sets of processes, or messages are routed through intermediate processes to their intended destinations. In this paper, we introduce a new programming abstraction — called canister communication — which permits the direct specification of global communications composed of multiple point-to-point message transmissions. Canister communication is a simple extension to existing message-based environments. Its use allows increased programming support for parallel computation throughout the software life cycle, including the code specification and debugging phases.

1. Introduction

It is the interprocess flow of data and control that distinguishes parallel from sequential computation. For massively parallel algorithms designed for tightly-coupled, nonshared memory architectures, this communication is often structured: streams of related data are pipelined along sequences of channels, values are broadcast to (and aggregated from) sets of processes, or messages are routed through intermediate processes to their intended destinations. In each case, the communications are best understood not as isolated point-to-point message transmissions, but as components of *global patterns of communication.*

These patterns are fundamental to our understanding of parallelism but they are not supported by existing parallel programming environments. In this paper, we introduce a new programming construct for global communication patterns. We begin with our motivation.

2. Motivation

Tightly coupled, massively parallel computations are best understood in terms of patterns of interprocess flow of data and control. In systolic applications [5,10], for example, data is pipelined along logical paths in fire-brigade fashion with messages passing from one process to the next over connecting channels. Figure 1 shows a systolic band `matrix multiplication` algorithm [11] with three such paths: the A matrix is carried from left to right, the B matrix is carried from top to bottom, and the resultant C matrix is carried diagonally from the bottom right to the top left. Each process reads an incoming value from each matrix, computes an inner product ($c = c + a \cdot b$) modifying the C value, and passes the values on. Figure 2 shows the same processes interconnected with different logical paths to implement a `lower triangular systems solver` [11]; in this case, all but the leftmost process in this array perform the innerproduct of band `matrix multiplication`, while the leftmost process does a pivot operation. Figure 3 shows a systolic `transitive closure` algorithm which computes the transitive closure of a matrix using a processor array of the same shape. It has two cyclic paths, one rotating horizontally and the other vertically; matrix values flow along these paths and are updated when they pass through processes at their "home" positions. In each of these examples, values flow through processors with specific access rights along prescribed, logical paths.

Nonsystolic algorithms also require logical paths. The `parallel prefix` algorithm for MIMD architectures [6], for example, accumulates results using a logical binary tree. On each iteration, all participating processes execute the same code, receiving from their

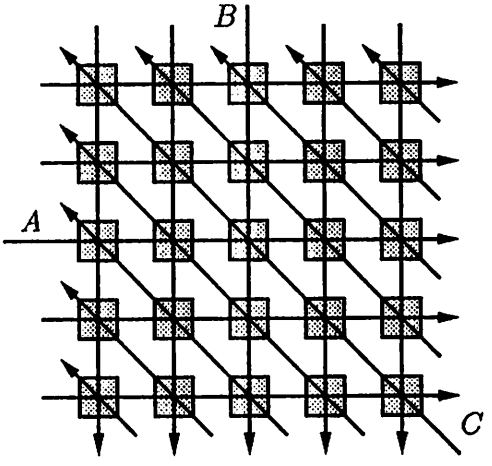


Figure 1: Paths of communication in band matrix multiplication of Kung and Leiserson [11].

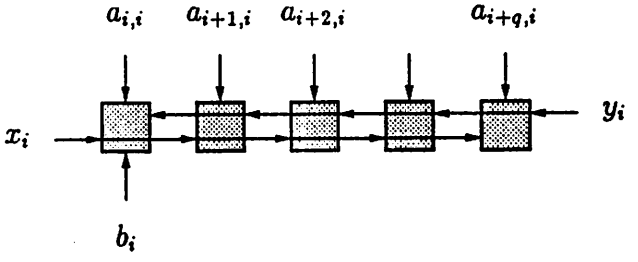


Figure 2: Solution of lower triangular systems [11] involves paths for $a_{i,j}$, x_i, y_i and b_i.

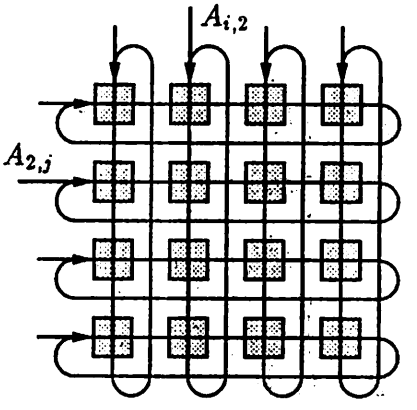


Figure 3: Cyclic data flow in a `transitive closure` [9] algorithm.

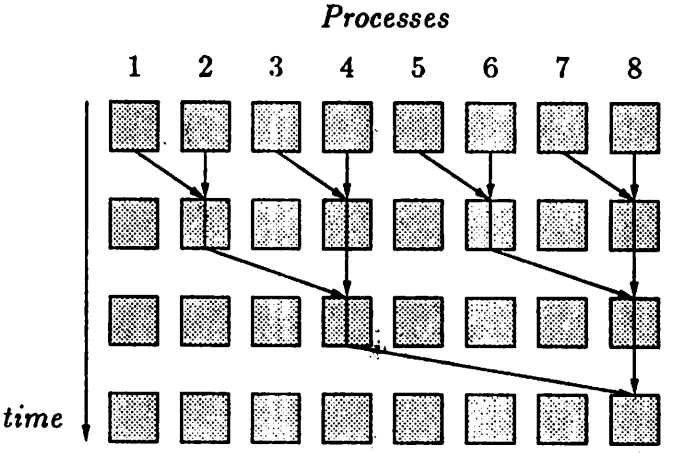


Figure 4: The path of the `sum-prefix` operation [6] is logically a complete binary trees.

children and sending to their parents along this path as shown in Figure 4. Such fan-in is also seen in itineraries for selection algorithms — used, for example in database computations — where a number of inputs are combined to form a single result as in Figure 5(a). Fan-out is seen in itineraries for broadcast and branch-and-bound algorithms as in Figure 5(a).

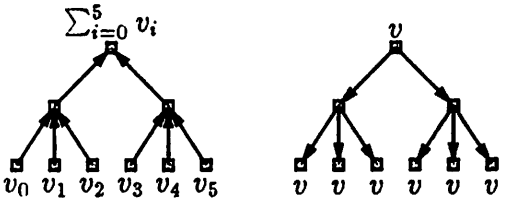


Figure 5: Fan-in and fan-out are demonstrated in the paths which implement competition algorithms (a) and broadcast algorithms (b), respectively.

In a `Jacobi iteration` for solving PDE's [13], processes repeatedly broadcast their current value to neighboring processes and then average the values they receive from those neighbors. Each message is sent on an export path and received on an import path as shown

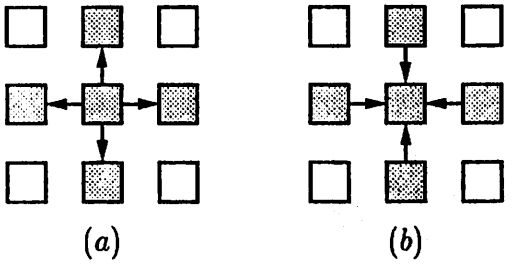


Figure 6: The itinerary for (a) exporting, and (b) importing values in Jacobi's PDE approximation technique [13] from the perspective of the center process.

in Figure 6. Another instance of messages persisting across sequences of transmissions is found in the `palindrome generation` algorithm of Finkel[4] where each process computes a small number of digits in an approximation of a large palindrome. Each iteration has two phases as shown in Figure 7; in the first, values are added across vertical channels, and in the second, waves of data *ripple* through the array propagating carry values, signals for rebalancing the computation, and the test for palindrome symmetry. In each of these examples, the logic of the algorithm is independent of the specifics of communication such

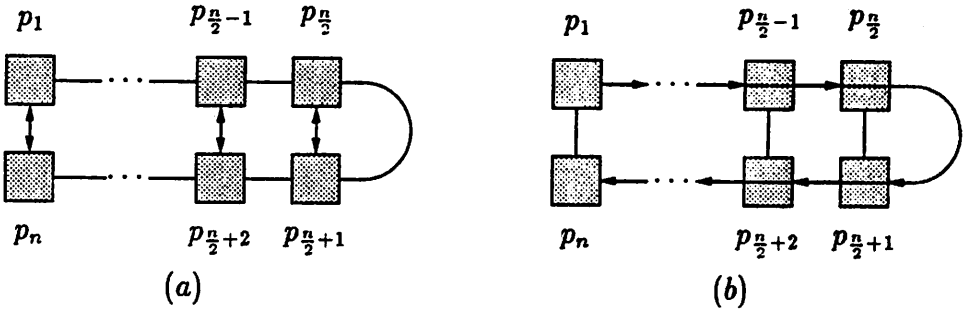


Figure 7: The palindrome generation algorithm [4]: the addition is carried out along vertical channels (a) and waves of data *ripple* through the array of processes (b).

as degree of fan-in or fan-out, multiplexing of channels, or position within the processor array.

Logical communication patterns can also serve as virtual circuits in which a message, traveling between logically adjacent processes, must traverse a succession of channels in a sparse network. This may be due to limitations in the connectivity of the hardware or it may be inherent in the algorithm itself. Examples of hardware induced routing occur in the LU Decomposition shown in Figure 8 and in the support of bitonic sorting and fast Fourier transform on square meshes shown in Figure 9. The Wavefront Array Processor implementation of LU decomposition [11,12] uses different communication paths in processing the pivot value, generating L-values, and generating U-values. In addition, it uses a diagonal shift of the matrix in successive reductions of the problem; on a square mesh, this shift corresponds to two shifts, one vertical and the other horizontal as shown in Figure 8(d'). The bitonic sort and the FFT use butterfly interconnections [15] which must be routed along mesh connections in multihop patterns as in Figure 9. Even in systems that provide automatic routing, the programmer may want to code it directly for efficiency [7]. An example of an algorithm with inherent routing is the Linear System Solver[14], shown in Figure 10; it requires complex routing to correctly interface each of several logical stages of the algorithm.

In each of these examples, global communication patterns are fundamental to our understanding of the algorithm and yet, no existing programming environment supports their direct specification. Existing environments allow specification of only point-to-point message transmissions, leaving communication behavior underspecified and error prone.

In this paper, we introduce a new programming abstraction, called *canister communica-*

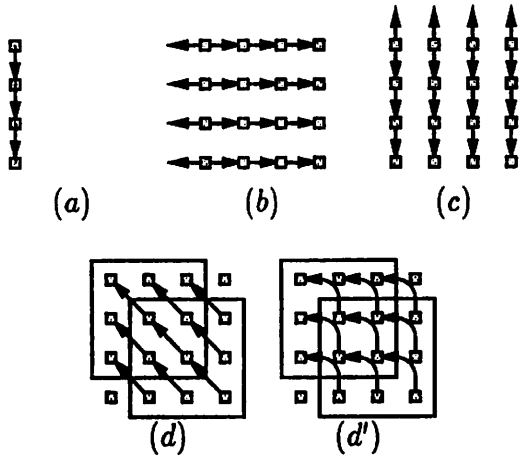


Figure 8: Wavefront Array Processor implementation of LU decomposition: process the pivot value (a); generate L-values (b); generate U-values (c); and perform successive reductions of the problem by logical shift of the matrix diagonally (d), which, on a square mesh, corresponds to two physical shifts (d').

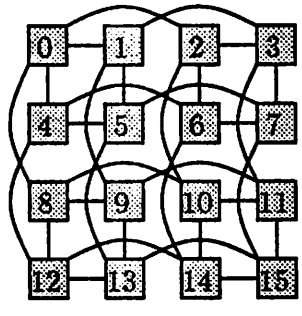


Figure 9: Butterfly networks are useful in implementing `bitonic sort` and `fast Fourier transform`. When such algorithms are embedded in a mesh processor, all channels must be routed along mesh connections.

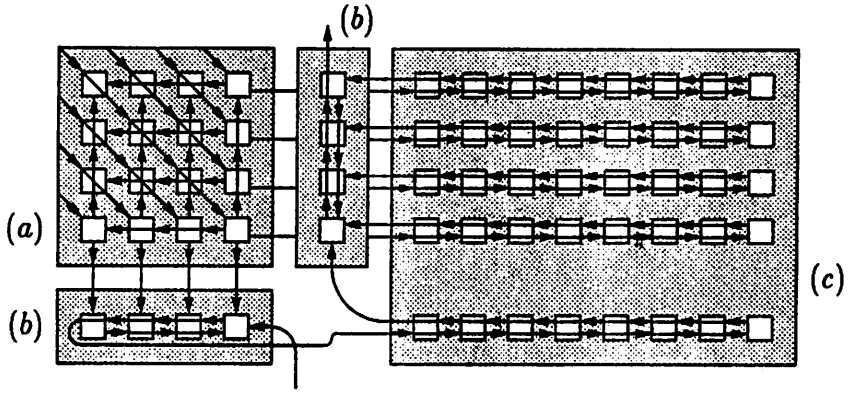


Figure 10: Paths supporting the solution of a system of system of linear equations. Paths result from the composition of `LU decomposition` (a), two lower triangular systems solvers (b), and stack of processors for intermediate matrix values (c).

tion, that enables programmers to specify global communication paths directly, narrowing the conceptual gap between the design of an algorithm and its implementation. Canister communication is a straightforward extension to existing message-passing environments. It allows the programmer to specify three important pieces of information related to the passing of messages in his algorithms: (1) the *type* of the data, (2) the *path* of the data through the system, and (3) each process' *local access rights* to the data along that path. Since many programming errors stem from the underspecification of message behavior, the use of canister communication helps the programmer avoid communication-related errors. By separating the details of communication from the logic of the algorithm, it often leads to concise specification of process code. In addition, canister communication provides a basis for several parallel debugging techniques.

In the next section, we formalize the notion of canisters. In Section 4, we demonstrate the role of canister communication in program specification and in Section 5, we discuss its role in parallel debugging; in Section 6, we offer some conclusions.

3. Canister Communication

The term *canister* is motivated by the pneumatic devices used for transactions in old stores and banks. For our purposes, a *canister* is a logical container for messages that traverses a prescribed path of point-to-point transmissions, called an *itinerary*. Each

canister is created, filled, transmitted, emptied, and destroyed in a manner consistent with its itinerary.

We first discuss itineraries.

3.1 The Itinerary: Expected Message Behavior

An itinerary is a path through a process array together with the type of data that it is to carry and the permissible accesses to that data. It is given as a node-labeled, directed graph in which nodes represent processes and edges represent communications. Because processes may play the role of several agents in the transmission of a message along its itinerary, a process may have several identities, or *aliases*. As a result, we label each node with (`process-name`,`itinerary-alias`) pair, which uniquely identifies the purpose of the node in that itinerary. Nodes are further labeled with a set of access rights that determine which canister operations — create, destroy, read, write, and view — are permissible. The possible orderings of such accesses are given in Figure 11.

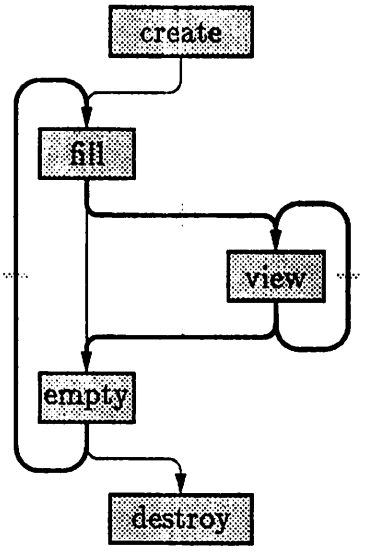
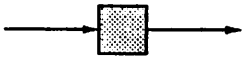
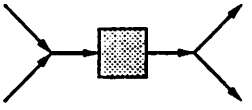


Figure 11: The possible access orders of canisters. Cuts indicate points where logical message transmissions must occur.

The simplest itineraries are lines which enter and exit distinct processes at most once:



If an itinerary enters (exits) a node in the graph more than once, multiple inputs (outputs) are assumed to be indistinguishable to the process:



Where this results in fan-in, values on all entering channels are combined with a merge function; where it results in fan-out, copies of the departing value are broadcast along each edge. We propose two merge functions: `SelectAny` which returns message chosen at random from those channels with messages available; and `SelectAll` which returns a set of messages, one from each incoming channel. Formally, an itinerary is defined as in Figure 12.

While itineraries describe global communication, their implementation is distributed throughout the system with each participating process storing its own local view of the path.

3.2 The Canister: A Message Wrapper

The *canister* provides a reusable logical wrapper for messages which is preserved across multiple transmissions as depicted in Figure 13. It is created in association with an itinerary, carries messages from point-to-point, and is eventually emptied and destroyed. Processes may read, modify or view the contents of the canister in accordance with the access rights mandated by its itinerary. All itineraries in the `band matrix multiplication` of Figure 1, for example, allow access to the values at each supporting process but the diagonal itineraries in the `LU-decomposition` of Figure 8 do not allow access by intermediate processes.

Each canister keeps track of its status with respect to its itinerary and this, in conjunction with the local processor views of that itinerary, is used to regulate access to message values. Canister communication thus makes it possible to ensure that communication is consistent with specified global patterns.

4. The Use of Canister Communication in Program Specification

Canister communication has been implemented as part of the Simple Simon[3] programming testbed. We use a textual database language (TDL) that allows the programmer to describe and annotate the communication structure associated with his algorithm. Itineraries are supported by TDL's path directive, which establishes the relation between

An itinerary has three components:

1. A domain M — called its *type* — from which messages are selected. A unique null message, $\perp$, is not in any domain, and is the message returned by a non-blocking read when no message is available on a channel.

2. A directed graph, G, where each node represents a process and each edge represents an adjacency in the process structure graph. (Processes may be self-adjacent.)

3. A labeling of the graph G, as follows:

   - Each node is uniquely labeled with a `(process-name,itinerary-alias)` pair.
   - Each node with outdegree $d > 0$ is labeled with an associated broadcast function

     $$broadcast^d : m \mapsto (m, m, ..., m)$$

     which generates a copy of the message for each exiting edge. (Nodes with no fan-out are labeled with the identity function $I = broadcast^1$.)
   - Each node with in-degree $d > 0$ is labeled with an associated merge function that combines available messages. We propose two such functions,

     $$SelectAll^d : M^d \rightarrow M^d$$

     $$SelectAny^d : (M \cup \{\perp\})^d - \{\perp\}^d \rightarrow M$$

     (In either case, we assume that processes are unable to distinguish between incoming channels.)
   - Each node is labeled with a subset of the access rights $\{read, write, view, create, delete\}$ to canisters and the messages they contain. *create* and *delete* respectively support sources and sinks for canisters in the itinerary, while *read*, *write* and *view* provide destructive read, write, and nondestructive read access respectively to messages in a canister.

Figure 12: The formal definition of an itinerary.

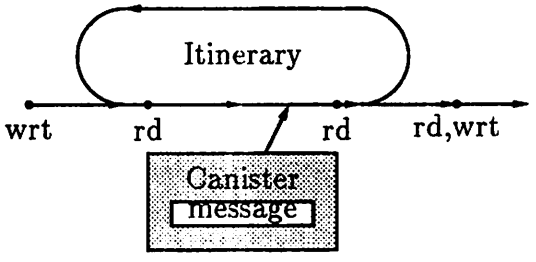


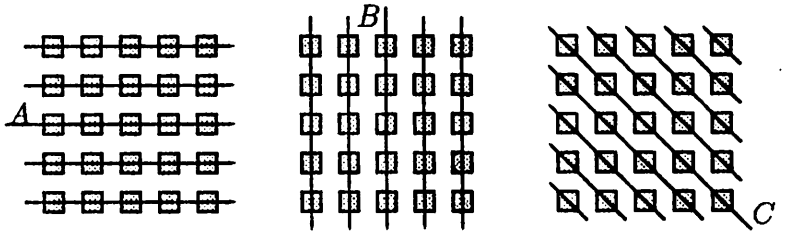
Figure 13: The canister and its associated itinerary.

channel connections and paths, and defines access rights and local aliasing. Canisters are implemented as a library augmenting the standard Simon message passing mechanisms.

This implementation was intended as a prototype; the interest was not in questions of elegance or syntax but in providing a testbed for experimenting with the concept of canisters. In particular, a graphical interface to the database (currently under development) will provide for more natural specifications of communication structures and itineraries.

We present here a few examples illustrating the utility of canister communication,giving only the relevant portions of the code.

Example 1. *Canisters support pipelining of data.* Because of the simplicity of the pipelines in band `matrix multiplication`, the canister-based code is very similar to the traditional message-based code. The specification of itineraries for the A, B, and C matrices, however, makes it possible to detect mismatched communications (for example, a message read as an A value but sent as a B value) that can not be detected in the message-based case. Itineraries for each of the matrices are



while the code supporting matrix transactions on these itineraries is as follows:

```
matmult() {
    /* declare canisters */
    CanTypeDecl(double,ACan);
    CanTypeDecl(double,BCan);
    CanTypeDecl(double,CCan);
      ...
    for (...) {
        CanPut(CanView(CanGet(ACan,APath),a)); /* process A value */
        CanPut(CanView(CanGet(BCan,BPath),b)); /* process B value*/
        CanEmpty(CanGet(CCan,CPath),c);
        CanPut(CanFill(CCan,c + a*b)); /* update C value */
    }
}
```

Example 2. *Canisters support reduction and broadcasting of data.* A systolic tree summation algorithm performs parallel summation of vectors that are piped in from the leaves of the tree. Sums appear at the root. Since each process sends the sum of its subtree to its parent for inclusion in the sum at the next level, the itinerary is a complete binary tree. The fact that the itinerary has the same topology as the underlying communication structure indicates that all channels support the same logical operation — information that cannot be gleaned from low-level message passing operations.

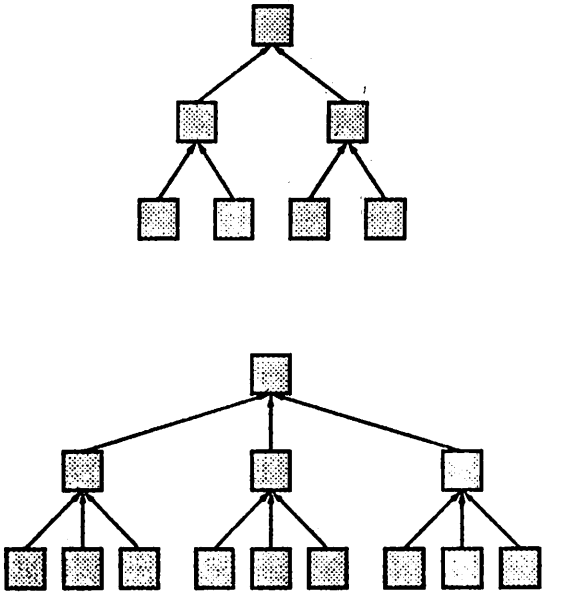
Code for the internal processes in the tree summation appears as:

```
int sumfun(a,b) { return a+b; }

tree() {
    CanTypeDecl(int,SumCan);

    CanGet(SumCan,TreePath,sumfun); /* aggregate incoming values */
    CanPut(SumCan); /* pass result on */
}
```

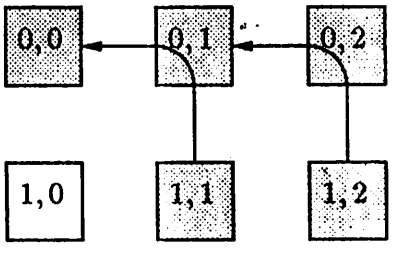
The code for the leaves and root would be different, but similar, since they must create and destroy canisters for the itinerary. If the external I/O interface were extended to support canister communication, however, all of the code could be specified with this single code body. In fact, this code body would work for trees of any degree



and it would work for unbalanced trees. Canisters thus allow the user to ignore irrelevant specifics of communication such as degree or position within the processor array or relation to external I/O boundaries.

Example 3. *Canisters support routing of data.* Many algorithms for the Wavefront Array Processor can be directly converted to run in the Simple Simon environment. Simon code bodies correspond to instances of Kung's *local view* of the WAP implementation. The monolithic code for the WAP processor considers computation in light of global communication *wavefronts*, a communication abstraction not supported in other programming environments. However, in `LU-decomposition` describing communication patterns using itineraries makes multiplexing of channels, and routing of the matrix during the reduction process more straightforward. Implementing the paths depicted in Figure 8, significantly increases the chances of detecting communication errors in these algorithms.

The `Shift` path illustrates the itinerary required to support the reduction phase of the WAP `LU-decomposition` algorithm.



Here, process 0,1 participates in the path as an intermediary under the alias Pass and as a recipient under the alias Shift. The process accesses these points of the itinerary with logically distinct operations.

```
lud()
{
    CanTypeDecl(double,SCan);
     . . .
    /* Shift the matrix: */
    CanPut(CanGet(SCan,Pass));
    CanDestroy(CanEmpty(CanGet(SCan,Shift),a));
     . . .
}
```

A canister created and sent from process 1,2 along the shift itinerary is forward by 0,2 using the pass view, and received by 0,1 as part of the shift. The alias encodes local logic while the itinerary encodes global flow.

For these examples, it can be seen that canister communication supports code specification in at least two ways: it often leds to more uniform code bodies and thus more concise programs because it separates the logic of the computation at a node from the details of communication; and it reduces communication errors because it proscribes violations of global communication patterns.

5. The Use of Canister Communication in Debugging

Parallel programs are difficult to debug both because of their inherent complexity and because they are not amenable to sequential debugging techniques. Appropriate treatment of communication can make debugging simpler. In particular, we have found that canister communication can help in at least two ways: it defines the intended patterns of message traffic and it provides a vehicle for attaching special purpose tags to messages. We briefly describe these uses.

5.1 Recognizing Patterns in Communication

The complexity of a parallel program can be overwhelming when approached from a low-level perspective and thus many parallel debuggers use some form of behavioral abstraction. In pattern-oriented debuggers such as those of Bates[2] or Hough and Cuny[8],

users model expected behavior by defining abstract events in terms of sequences of system defined primitive events; the debugger recognizes these patterns within the event stream and reports on their occurrence. This provides a powerful mechanism for coping with complexity — abstraction — but it also requires explicit model building which can involve considerable effort on the part of the user. For tightly coupled, massively parallel systems, however, the modeled behavior often consists of the same global communications supported by canisters. Thus canister specifications could be compiled into high-level event definitions, making them available to the debugger directly without further modeling by the user. This would automatically provide an appropriate level of abstraction for many debugging activities.

5.2 Radioactive Tagging

Communication errors in parallel programs often result in computations on the wrong values. One possible method for detecting such errors is to *radioactively tag* values in order to trace their influence:

> Initially, data values would be tagged. As execution proceeds, tagged values would *contaminate* any computations in which they were used with intermediate values inheriting the tags of the values used to produce them in a dataflow manner. Results could then be inspected for appropriate tags.

Radioactive tagging could be enormously helpful, but it would be prohibitively expensive to implement.

Canisters provide the basis for an efficient approximation: the canister *itself* conveys tags which accumulate in participating processes. Both processes and canisters are labeled by sets of tags which obey the following rules:

1. Each canister is tagged at creation by the user, perhaps based on initial contents.
2. Each canister read or created at a process, appends its tag set to the tag set of the process.
3. Each canister sent from a process is retagged with the tag set of the transmitting process.
4. Merged canisters receive the union of tag sets, while duplicated canisters receive copies of tag sets.

Tagged canisters would be useful in debugging the `band matrix multiplication` above. If the canisters transmitting A values were tagged, for example, the shaded processes in Figure 14 should show contamination from values entering in the middle row.

If, in addition, the canisters carrying B and C values are tagged, examination of the tag lists of exiting C canisters would show whether or not appropriate terms are being computed (allowing us to detect many initialization errors which result in incorrect pairings of values). A similar technique would enable us to determine, for example, whether all pro-

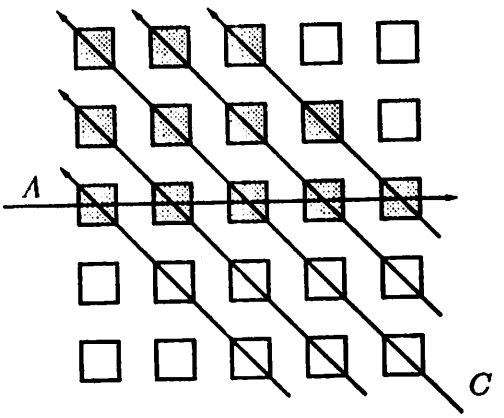


Figure 14: Contamination in `band matrix multiplication`: shaded processes should show contamination by canisters carrying A values entering in the center row.

cesses contribute in a `tree summation` or whether the generation of L-values is proceeding correctly in an `LU decomposition`.

6. Conclusions

We have proposed a communication abstraction based on the expected behaviors of message structures which persist along multihop paths. With this mechanism, programmers are able to describe an algorithm more lucidly, separating the logic of the algorithm from the low level details of the communication. Canister communication also provides debugging abstractions which can be used directly by a pattern-oriented parallel debugger.

We have implemented canister communication in the Simple Simon environment, using textual descriptions of itineraries. This is quite cumbersome and we expect to replace it with a graphical interface using a grammar-based graph editor[1]. We also expect to extend the implementation to automatically generate high-level communication event definitions for use by the Belvedere parallel debugger. Finally, since our communication patterns are

currently limited to those determinable at compile time, we are investigating methods for supporting dynamically initiated, but statically structured communication patterns.

Acknowledgements We would like to thank John Hagerman for his contributions during early discussions of the canister concept.

References

[1] Duane A. Bailey. *Specifying Communication for Massively Parallel Ensemble Machines.* PhD thesis, University of Massachusetts, Amherst, Massachusetts, 1988.

[2] Peter C. Bates. *Debugging Programs in a Distributed System.* Technical Report 86-05, University of Massachusetts at Amherst, January 1986.

[3] Janice E. Cuny, Duane A. Bailey, John W. Hagerman, and Alfred A. Hough. *Simple Simon Programming Environment: A Status Report.* Technical Report 87-22, University of Massachusetts at Amherst, March 1987.

[4] Raphael A. Finkel. *Large-grain parallelism — Three case studies*, pages 21–63. *Scientific Computation Series*, MIT Press, Cambridge, Massachusetts, 1987.

[5] L. J. Guibas, H. T. Kung, and C. D. Thompson. Direct VLSI implementation of combinatorial algorithms. In *Caltech Conference on VLSI*, pages 510–525, January 1979.

[6] W. Daniel Hillis and Jr. Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12), December 1986.

[7] Ching-Tien Ho and S. Lennart Johnsson. Distributed routing algorithms for broadcasting and personalized communication in hypercubes. In *1986 International Conference on Parallel Processing*, pages 640–648, August 1986. Binomial trees.

[8] Alfred A. Hough and Janice E. Cuny. Belvedere: prototype of a pattern-oriented debugger for highly parallel computation. In *1987 International Conference on Parallel Processing*, pages 735–738, 1987.

[9] H. T. Kung. Let's design algorithms for VLSI systems. In *Caltech Conference on VLSI*, pages 510–525, January 1979.

[10] H. T. Kung. Why systolic architectures? *Computer*, 15(1):37–46, January 1982.

[11] H. T. Kung and C. Leiserson. Systolic arrays (for VLSI). In Carver Mead and Lynn Conway, editors, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachusetts, 1980.

[12] Sun-Yuan Kung, K. S. Arun, Ron J. Gal-Ezer, and Bhaskar Rao. Wavefront Array Processor: language, architecture, and applications. *IEEE Transactions on Computers*, C-31(11):1054–1066, November 1982.

[13] Michael J. Quinn. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill, New York, 1987.

[14] Lawrence Snyder. Introduction to the configurable highly parallel computer. *Computer*, 15(1):47–56, January 1982.

[15] C. D. Thompson and H. T. Kung. Sorting on a mesh-connected parallel computer. *Communications of the ACM*, 20(4):263–71, April 1977.