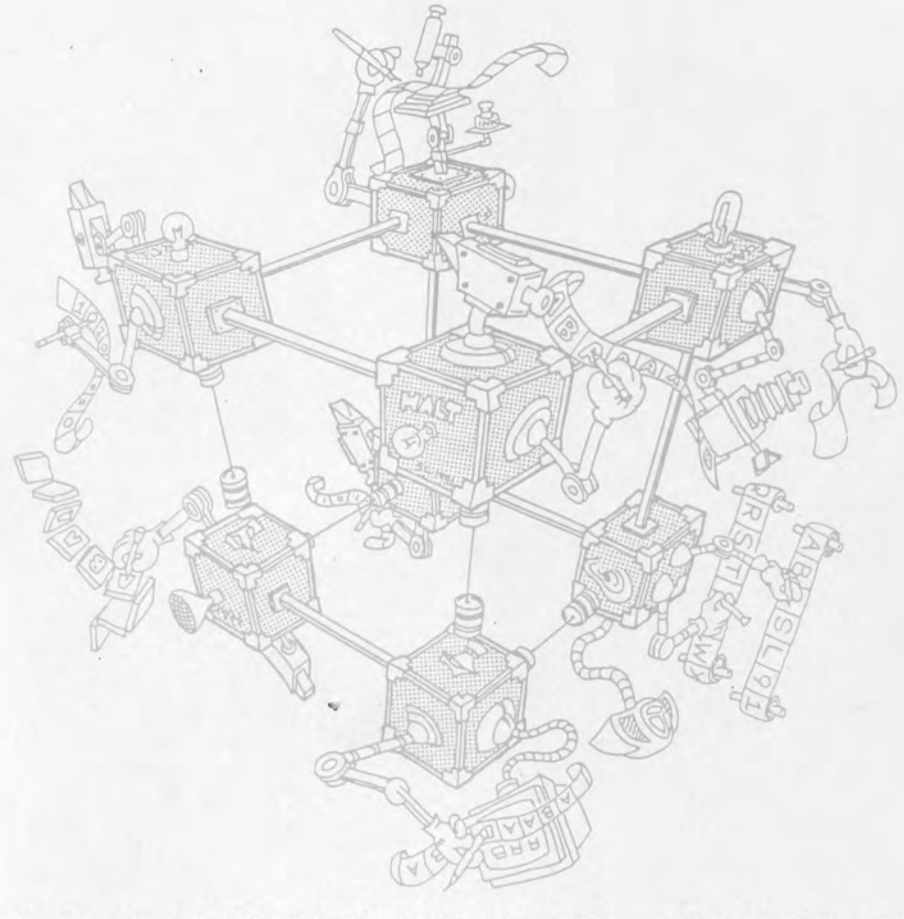


A VISUAL PROGRAMMING TOOL FOR FORTRAN D

Cherri M. Pancake
Prasanna K. Kondapaneni
Christopher Ward

92-60-23

Oregon State University
Department of Computer Science
Corvallis, OR 97331

A Visual Programming Tool for Fortran D*

Prasanna K. Kondapaneni
Technology & Communications Systems
prasanna@tcs.com

Cherri M. Pancake
Oregon State University
pancake@cs.orst.edu

Christopher Ward
Auburn University
wardc@eng.auburn.edu

December 7, 1992

Abstract

Visual Fortran D (VFD) is a graphical tool to assist parallel programmers in specifying data distributions. Its target is Fortran D, an extension to Fortran77 or Fortran90 which supports data parallelism. VFD provides an intuitive framework where the user employs simple, fast graphical manipulations to specify how data is to be organized for distribution across multiple processors. The corresponding Fortran D statement is generated automatically from the graphical representation and displayed alongside it. Initial experimentation by users indicates that VFD improves the accuracy of data distributions. The ability to observe how a specification statement varies as the graphical representation is changed appears to make VFD a useful tool for teaching Fortran D concepts well.

1 Introduction

Parallel scientific programs are difficult to specify and comprehend because of both the volume of information and the lack of clear relationships between program code and the problem being addressed [27]. Programming language notation — particularly the Fortran dialects used by most scientific programmers — tends to obscure even simple operations on data structures such as three-dimensional arrays, trees, or meshes.

*The work described here was carried out at the Department of Computer Science and Engineering, Auburn University, and was supported partially by a Joint Development Study with IBM Corporation.

Those operations are easier to understand when diagrams are employed. The human visual system can capture two- or three-dimensional images much better, and at a much faster rate, than text, which must be processed linearly [30, 34, 35]. Moreover, diagrams are more likely to reflect the nature of problem entities than is formalized text notation. These principles are the rationale behind recent developments in visual programming and algorithm animation (e.g., [1, 5, 22, 4]). In this paper, we demonstrate how the concepts of graphical languages can be extended to the world of Fortran. Recent efforts to standardize parallel Fortran have focussed on data-parallel language extensions [10, 14, 17, 6]. The programmer annotates a serial Fortran program to indicate when multiple processors should be applied and how program data should be partitioned among the processors. Visual Fortran D (VFD) offers a framework for developing the specifications graphically.

VFD is an interactive, object-oriented tool written in C++ and based on the X Window System platform and OSF/Motif widgets [19]. Program arrays, Fortran D decompositions, and physical processors are represented as graphical objects. They are manipulated by the user to specify how data is to be distributed. From those graphical specifications, the tool generates Fortran D annotations which are inserted into the original Fortran program. The Fortran D compiler analyzes the annotations and parallelizes the program, generating calls to a message-passing library [13].

This paper describes VFD and how it facilitates the process of specifying Fortran D data distributions. Section 2 outlines some of the recent efforts in graphical tools from which our work is derived. It is followed by a brief discussion of the Fortran D language. Section 4 describes how VFD allows the user to generate those specifications via graphical manipulations of arrays and other objects on the screen. A final section draws conclusions about the usefulness of the work and how it might be applied to other languages and program development activities.

2 Related Work

A number of recent efforts have addressed the use of graphical techniques to streamline parallel program debugging (see [28] for a list of publications in this area). Although most of the tools are still prototypes, initial results from practical experiments indicate that visualizations can be of considerable help in spotting anomalous interprocessor communications [15], race conditions in the access of shared data [20], improper boundary conditions in message-passing algorithms [32], improperly sequenced array updates in linear system algorithms [7], and other errors which would be difficult to detect in text-based systems. Similarly, graphical representations have been effective in isolating message bottlenecks or memory contention in computationally intensive parallel programs [18, 8, 21, 11, 12].

VFD is the first tool to support graphical specifications of data distributions for parallel programs. Since those specifications are expressed in terms of the program's declared arrays, our motivation and the VFD implementation are most closely related to work in matrix visualizations. Graphical representations of arrays have been used to facilitate parallel programming in several ways:

- to portray the behavior of parallel algorithms (e.g., [33])
- to help debug parallel programs that access arrays (e.g., [2, 8])
- to display performance characteristics (e.g., [18, 11])

From these experiences, it is clear that visualizations can be useful for interpreting both the contents of arrays and the patterns of access to the data storage locations they occupy.

Previous efforts have concentrated on the usefulness of graphics in reformulating large, complex data streams into a form which is quickly grasped by the user [25]. In the context of matrix visualization, it is most important that graphical portrayals reflect very closely the programmer's mental model of array structure and use [26]. A grid made up of individual cells arranged in row(s) and column(s) captures the intuitive notion of arrays and requires no explanation and no special learning on the part of the user. VFD exploits this simple representational framework, applying it to the task of visual programming.

Typically, programming languages languages are very high level, allowing the user to specify algorithms in a manner which is much closer to the problem at hand than do traditional programming languages (cf. [1, 29, 30]). This is the basic approach of VFD, although the graphical manipulations are used to generate textual augmentations to a textual program. In the taxonomy established by [1], we provide a *form-based paradigm*, whose basic graphical objects are simplified representations, enabling the programmer to omit or elide the more complex syntactic elements of Fortran D. Since the user must ultimately debug Fortran D source code, however, the VFD interface also displays the text of all statements generated from the visual specifications.

3 Overview of Fortran-D

Fortran D is a series of extensions to Fortran77 or Fortran90[10]. It is intended to support fine-grained data parallelism. The model of parallelism, commonly referred to as SPMD (Single Program, Multiple Data), defines a single thread of control, which is applied simultaneously to multiple elements of composite data structures, in this case arrays. The user begins with a serial Fortran program, and adds Fortran D statements to specify how data elements should be distributed across multiple processors in order to minimize the amount of data movement from one processor memory to another. He or she also indicates which portions of the program should actually execute in parallel, by enclosing them in a Fortran D `FORALL` loop (similar to `DOACROSS` or `WHERE` statements in other parallel languages). Fortran D also provides a selection of so-called reduction operations — such as sum, product, minimum, maximum, or boolean functions — which are applied simultaneously to a collection of data elements in order to generate a single scalar value. Using these statements, the Fortran D compiler generates the appropriate MIMD message-passing calls or SIMD controls needed to parallelize the application [13].

The syntax and semantics of `FORALL` are straightforward, and are applied with relative ease to most computationally intensive loops. It is the specification of data mappings which constitute the major effort in writing Fortran D programs. (In fact, those specifications occupy all but three pages of the Fortran D language definition.) Moreover, the conceptual model presented by the data specifications is complex and unfamiliar to most programmers. The VFD tool was originally conceptualized when the Fortran D definition document was observed to rely almost exclusively on diagrams to explain the meaning of language constructs [10]. The advisability of a graphical specification tool was confirmed during the first meeting of the High Performance Fortran Forum — a standards group whose proposed language is based in large part on the Fortran D effort — where even language and compiler experts resorted to sketches in order to clarify proposals for data distribution mechanisms. VFD, then, provides a faster, more intuitive framework for specifying how data is to be organized for distribution across multiple processors. As Fortran D is likely to be unfamiliar to the reader, we begin with a synopsis of the pertinent features.

The programmer specifies how arrays are to be mapped to processors via an indirect, two-step process. The first step is referred to as the *problem mapping* because it is intended to reflect the nature of the problem rather than the number or type of processors upon which the program will execute. The programmer defines abstract entities called `decompositions`, each representing "an abstract problem or index domain" [10]. In terms more understandable to the average Fortran programmer, decompositions allow the user to indicate how arrays will be accessed by the program (e.g., row-wise, column-wise, planar, by quadrant, etc.) so that data locality can be exploited.

Decompositions are defined with the `DECOMPOSITION` statement, which is similar to the declaration of an array, but allocates no storage:

```
DECOMPOSITION MESH(100,100)
DECOMPOSITION LINEAR(50)
```

Program arrays are then mapped to decomposition. Since all arrays mapped to a given decomposition are aligned with each other by the compiler, the decomposition abstraction provides an indirect way of expressing how arrays will relate to each other. The `ALIGNMENT` statement is the richest, syntactically and semantically, in Fortran D. For example, given two program arrays `A(N,N)` and `X(I)`, the possibilities include:

- Exact match of array to decomposition: `ALIGN A(I,J) WITH MESH(I,J)`
- Transposition of array with respect to decomposition: `ALIGN A(I,J) WITH MESH(J,I)`
- Align array at an offset with respect to decomposition: `ALIGN A(I,J) WITH MESH(I+1,J-2)`
- Alignment with an inter-element stride: `ALIGN A(I,J) WITH MESH(I,3*J-2)`
- Embed an array inside a larger decomposition: `ALIGN X(I) WITH MESH(I,2)`
- Collapse (ignore) one dimension of an array: `ALIGN A(I,J) WITH LINEAR(I)`
- Replicate an array in a larger decomposition: `ALIGN X(I) WITH MESH(I,*)`
- Bounded replication: `ALIGN X(I) WITH MESH(I,1:4)`
- Array ranges: `ALIGN A(I,J) WITH MESH(I,J) RANGE(*,2:N-2)`

These mechanisms may be combined arbitrarily.

The programmer must also decide what to do if the array size exceeds that of the decomposition in any dimension. Consider the following alignment:

```
ALIGN X(I) WITH LINEAR(I+2)
```

If `X` and `LINEAR` are of the same dimension, attempts to access the last two elements of `X` will overrun the places established by the decomposition. Fortran D's `OVERFLOW` clause establishes what will happen in this case:

- `OVERFLOW (ERROR)`, the default, will result in a runtime error.
- `OVERFLOW (TRUNC)` will return the same value as the last valid element of `X`.
- `OVERFLOW (WRAP)` wraps the values around, returning the first and second elements of `X`.

Clearly, the alignment syntax encompasses a broad range of subtleties that may be confusing to an inexperienced or infrequent programmer.

Once arrays have been mapped to the appropriate decompositions (note that they can be re-mapped dynamically at arbitrary points in the program code), the second step, referred to as the *machine mapping*, is specified. Here, the abstract decompositions are mapped to physical processors, thereby establishing how array elements will be distributed to processor memories during program execution.

The `DISTRIBUTE` statement provides this mapping. Three modes of distributions are supported in Fortran D:

- `BLOCK` partitions the array into block units for distribution across the processors.
- `CYCLIC` distributes elements round-robin style to the processors.
- `BLOCK_CYCLIC` partitions the elements into blocks, which are then distributed round-robin style to the processors.

The three modes may be combined in arbitrary ways, one per decomposition dimension. Moreover, each mode can be qualified by a number indicating the size of the partition unit. The following are examples of valid distributions for an array of rank two:

```
DISTRIBUTE A(BLOCK_CYCLIC,BLOCK(10))
DISTRIBUTE A(BLOCK(N$PROC),CYCLIC(10))
DISTRIBUTE A(BLOCK(2,N$PROC),BLOCK(5))
```

Finally, irregular distributions may be specified using an index map, in which each element indicates the processor to which the corresponding element of the decomposition should be mapped.

Fortran D is a somewhat fragile language, in the sense that program efficiency may vary dramatically depending on the programmer's ability to use the specifications wisely. Since simple transpositions or minor syntactic changes can have significant impact on performance — with no error or warning messages to alert the user – this is a programming situation which clearly could benefit from tool guidance.

4 Programming Fortran D Specifications Visually

Current graphical technology makes it possible to develop portable tools that can be used standalone or integrated with a programming environment or compiler. Graphics hardware and software are becoming cheaper every day, as ease-of-use becomes a dominating factor in purchasing decisions. In the arena of high-performance computing, most programmers are now familiar with the concept and use of graphical idioms, thanks largely to the expanding role of visualization in representing and interpreting scientific results [36, 9]. This situation encouraged us to explore the use of graphical frameworks to simplify and elucidate the specification of data distribution patterns.

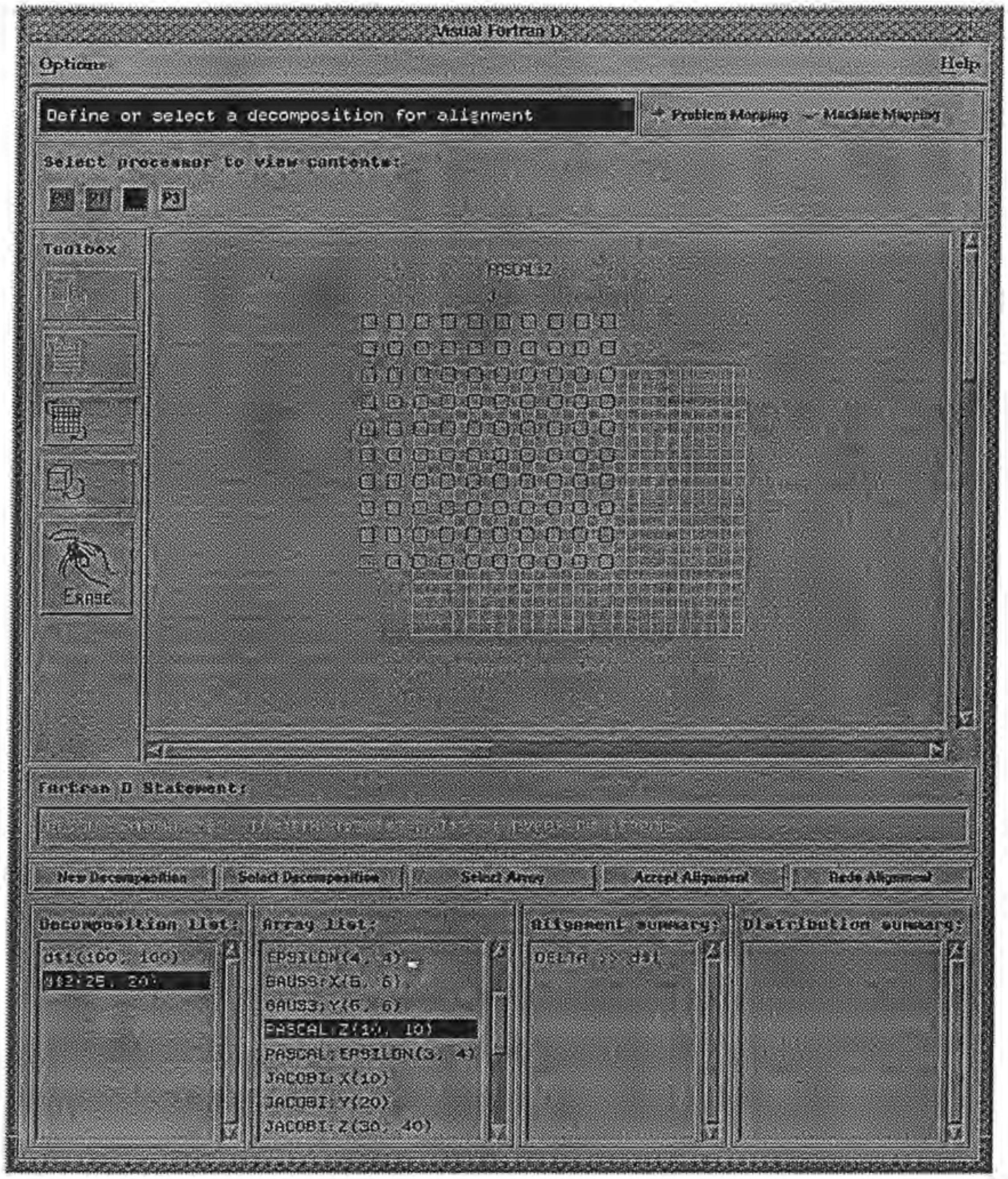
The appearance of VFD is shown in Figure 1. A *drawing area* occupies the central portion of the display. It is here that the user manipulates graphical entities representing arrays, decompositions, and processors. Those manipulations result in the generation of Fortran D statements, which appear in the *statement display* field just below the drawing area. Below this is a series of *buttons* used to select from among a number of available operations. The *scrollable lists* at the bottom of the display enumerate the arrays declared in the program, plus all Fortran D decompositions and alignments that have been defined in the current VFD session. The *menubar* at the top of the display offers additional options as well as access to an extensive help facility. A *message area* prompts the user with information on what to do next, and allows the user to select between problem and machine mapping modes.

Along the lefthand edge of the display, the user may choose *processor buttons* to see exactly how array elements will be distributed, or gain access to a *toolbox* for rotating or erasing the displayed objects. When needed, popup dialogs prompt the user for additional information:

- A decomposition definition dialog prompts for the rank, size, and name of the new decomposition.
- A stride dialog prompts for the distance at which elements of the selected array should be spaced.
- A collapse dialog determines which dimensions of an array should be flattened.
- A rotation dialog determines which two dimensions of a rank-3 array/decomposition should be displayed (and distributed).
- A processor allocation dialog prompts for the number and shape of the processor array.

The interface was designed to follow the logical sequence of steps that a programmer employs in writing Fortran D code. In the absence of a graphical tool, users typically draw pictures of their arrays or decompositions to help conceptualize how data should be distributed onto processors. `DECOMPOSITION` and `ALIGNMENT` statements reflect those sketched alignments. The step of mapping the decomposition to processors is typically also assisted, by a second set of sketches or by modification to the earlier diagrams. VFD automates the tedious and error-prone processes of drawing arrays and of translating mentally from the intuitive model to textual Fortran D. It also guides the user through the actions with a series of instructions displayed in the message area.

When the VFD application is started, it loads the array declarations from the user's source program. (The full implementation of the tool will rely on the companion compiler to supply this information.) The array specifications are inserted automatically into the array list area. As the user defines each decomposition or selects an array, VFD draws a rectangular grid representing the object; arrays are distinguished visually from decompositions by color. The user manipulates an array object directly, moving it into place over the appropriate decomposition object. He or she may choose to apply several optional operations to an array object, such as collapsing a dimension or applying stride(s). As the interaction progresses, VFD displays the `ALIGNMENT` statement textually; the field is updated continuously to reflect all modifications to the positions



```
Define or select a decomposition for alignment

Select processor to view contents:

Fortran D Statement:
[illegible]

Decomposition list:
d1(100, 100)
d2(25, 20)

Array list:
EPSILON(4, 4)
GAUSS:X(6, 6)
GAUSS:Y(6, 6)
PASCAL:Z(10, 10)
PASCAL:EPSILON(3, 4)
JACOBI:X(10)
JACOBI:Y(20)
JACOBI:Z(30, 40)

Alignment summary:
DELTA -> d1

Distribution summary:
```

Figure 1. Visual Fortran D display

of the graphical objects. To specify a distribution, the user manipulates the decomposition object, indicating what subportion should be positioned on the first processor. Based on this information, VFD automatically calculates what elements will be distributed to other processors, and updates the display to show the complete mapping as well as the text of the DISTRIBUTE statement. The procedures are described in more detail below.

4.1 Defining the Problem Mapping

Clicking on the "New Decomposition" button pops up a window requesting the user for the name, dimension, and rank to be used. The popup has buttons which support the OSF/Motif standard semantics [23]; that is, the user optionally can define multiple decompositions at once, cancel the entire operation, or obtain more detailed help directly from the popup. To minimize the amount of typing required, VFD automatically generates decomposition names (these can be overridden by simple editing of the text field). For the other parameters, the previous values are remembered and supplied as defaults, so that it is possible to define a whole series of similar decompositions with just a couple of user actions.

As each decomposition is defined, it is inserted into the scrollable list at the bottom of the display. (Recall that declarations of arrays were acquired transparently from the source program and inserted into the scrollable array list.) The user maps an array to a decomposition — specifying a Fortran D alignment — by selecting the appropriate array and decomposition objects from the listings. The two objects appear in the main drawing area, distinguishable by color and by a textual label providing the name of the array. By default, VFD aligns the two with an exact match; that is, the first element of the array overlaid on the first element of the decomposition. To change this alignment, the user simply drags the array object to the desired location over the decomposition. VFD accommodates user imprecision in superimposing the objects by snapping the array to the nearest grid point on the decomposition.

As the array is dragged, VFD automatically calculates the offsets for the ALIGNMENT specification from the relative position of the array with respect to the decomposition. This is recalculated whenever the array is moved on the screen, and the text field displaying the Fortran D statement is updated accordingly. There are two mechanisms for aligning large objects that extend beyond the viewing boundaries of the window:

1. Automatic scrolling: As the user approaches the boundary of the clipped window, the application moves the scrollbars and pans across the window, so dragging an array object in a large clipped window becomes easier. Since the current offsets are updated in the text field, the user can stop precisely at the desired location.

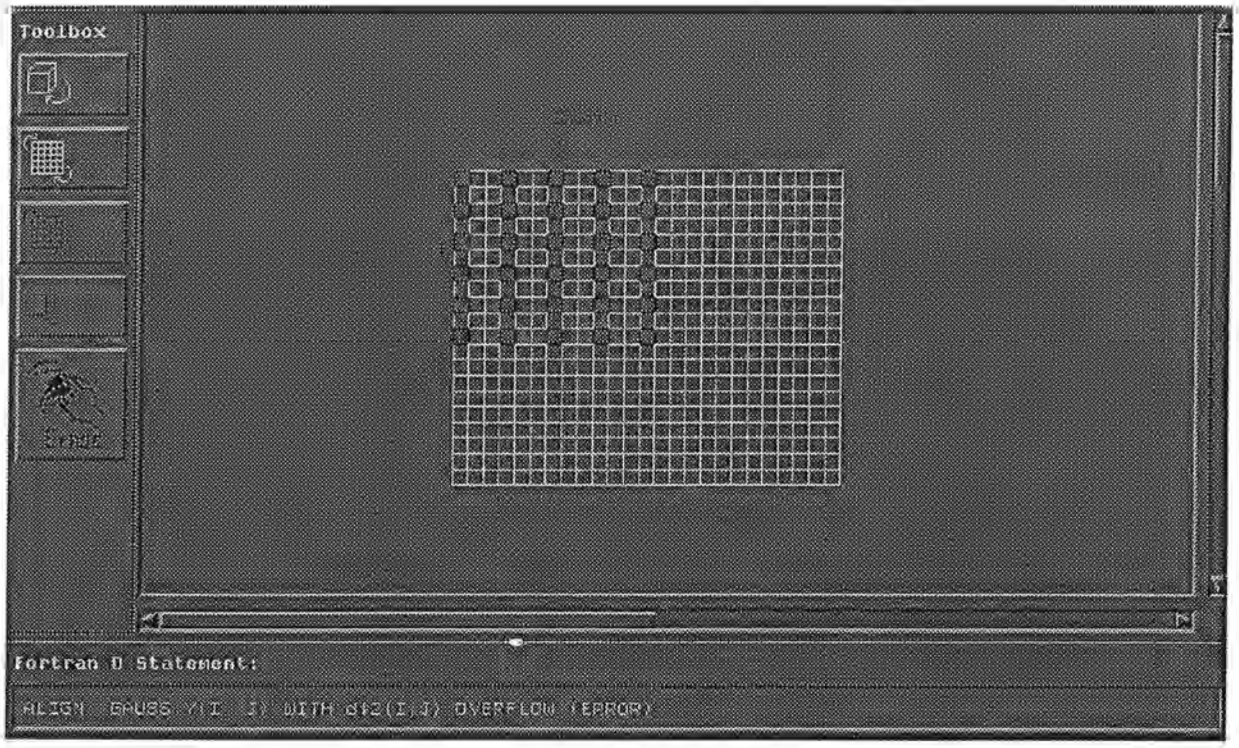
2. Scaled map: Optionally, a small map of the large scrolling area is displayed where the user can align scaled-down objects quickly. This mechanism offers a clear picture of the relative sizes and locations of the desired alignment. Fine tuning can then be done using the full-size objects.

Permutations are specified by applying the toolbox functions to rotate the objects on the display. Alignment strides are supported through the stride popup dialog, available from the Options menu. The user simply modifies the stride (the default is 1) in one or more directions, and verifies the operation visually by observing how the display changes. In Figure 2, for example, an array of size (5,6) has been aligned with a decomposition (25,20), and the stride adjusted to be 3x2. The visual display makes it easy to detect and correct errors in stride specification.

If the user wishes to collapse particular dimensions (i.e., ignore them in an alignment), this can also be accomplished via a menu option and verified graphically. The three overflow options are supported graphically as well. If the `ERROR` option is in effect, an `X` is displayed in each element outside the decomposition to emphasize the fact that accessing such an element will be an error. Similarly, `WRAP` is displayed by wrapping the edge elements around the decomposition, while `TRUNC` visually places all elements outside the boundaries into the last element of the decomposition. Alignments may be redisplayed and modified at any point during the VFD session.

4.2 Defining the Machine Mapping

When the user switches from problem- to machine-mapping mode, the button labels change to reflect the new operations which are available. (This step is consistent with the Fortran D model,



Fortran D Statement:

```
ALIGN  GAUSS_Y(I, J) WITH d12(I,J) OVERFLOW (ERROR)
```

Figure 2. **Problem mapping using alignment strides**

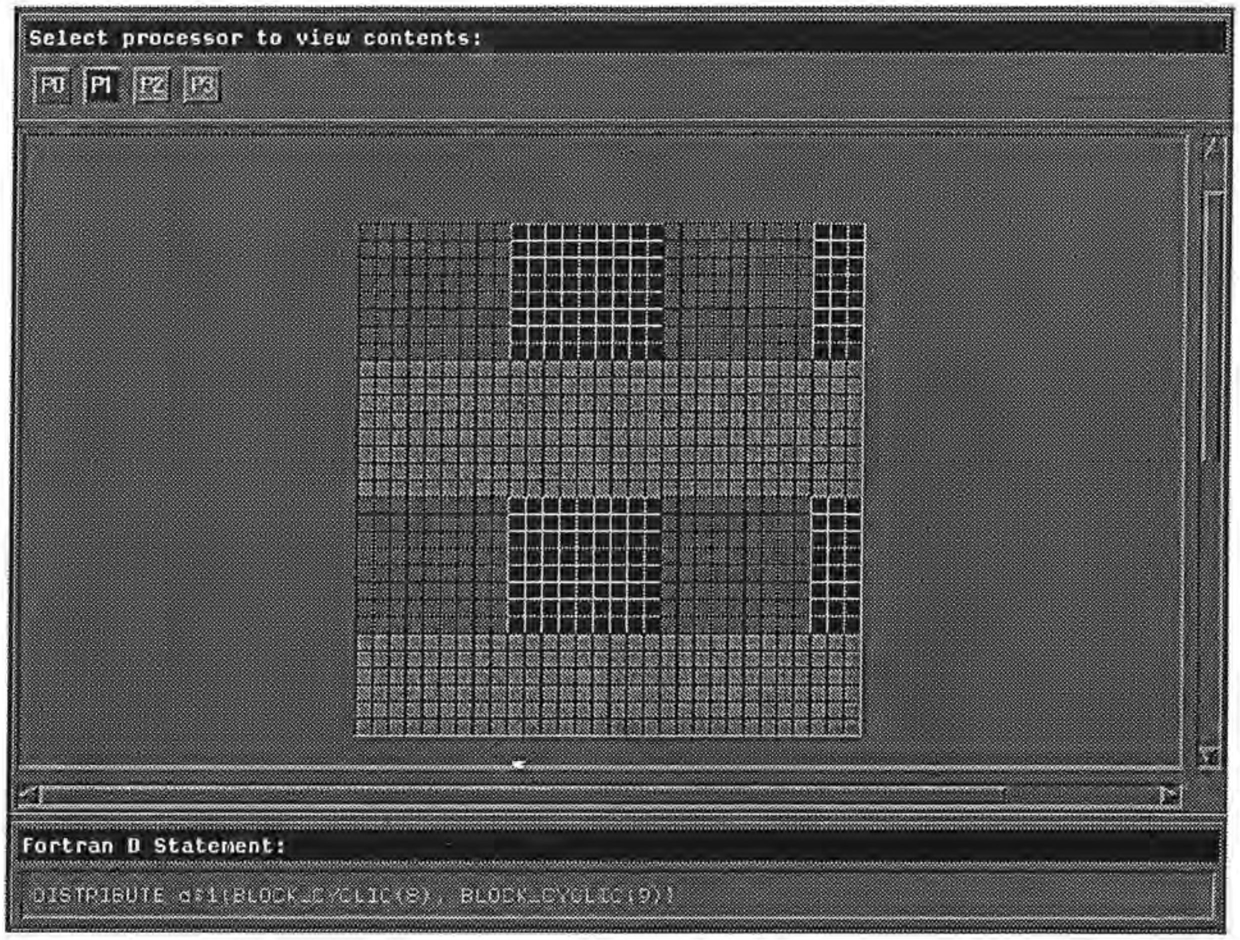
which represents the mappings as separate operations.) In machine mode, the user cycles through the decompositions, specifying how each is to be distributed across the processors.

The number and topology of the processor array is first specified via a popup dialog. The user then selects the decomposition to be distributed and it is drawn on the main panel. Although Fortran D provides three types of distribution, BLOCK and CYCLIC distributions are actually special cases of BLOCK_CYCLIC.[1] Therefore, all alignments generated by VFD are of the BLOCK_CYCLIC form. This simplifies the operation considerably. Using a rubber-banding mechanism, the user drags a rectangular outline over the portion of the decomposition which should be distributed to the first processor in the array. Once again, the corresponding Fortran D statement is displayed in the text field.

To distribute a decomposition in BLOCK fashion, the user selects the first N/P elements. The subportion of the decomposition that is mapped to each processor is distinguished graphically by color, keyed to the color legend at the top of the display. This makes it relatively easy to determine if the elements are distributed evenly. To specify CYCLIC distribution instead, the user selects only the first element of the decomposition; remaining elements are then distributed round-robin to other processors. For BLOCK_CYCLIC distribution, the user would select the size of the block to be distributed in cyclic fashion. Figure 3 shows an example of a two-dimensional BLOCK_CYCLIC distribution.

The buttons used to represent the color key also serve to encapsulate information about the processors. Selecting a button displays a popup window presenting a scrolled list of all elements in the current decomposition that will be mapped to that processor. This can be used to verify that arrays have been distributed correctly. Once again, the user is free to re-define distributions at any convenient point in the session.

[1]BLOCK_CYCLIC and CYCLIC differ only in the size of the blocking factor. BLOCK_CYCLIC(m) divides a dimension into contiguous blocks of size m and assigns each block to a processor in round-robin fashion. BLOCK_CYCLIC becomes equivalent to CYCLIC when m is 1. Similarly, when we force m=N/P, the dimension is divided into exactly P blocks and each processor is assigned exactly once, precisely the same as in BLOCK distribution.



```
DISTRIBUTE d$1(BLOCK_CYCLIC(8), BLOCK_CYCLIC(9))
```

Figure 3. Machine mapping using `BLOCK_CYCLIC distribution`

5 Conclusions

Visual Fortran D represents an attempt to make the Fortran D programmer's job easier and more efficient. Initial experimentation by users indicates that VFD indeed improves the accuracy of data distribution by clearly portraying the abstract and machine mappings, and their interrelationships. What's more, the ease with which data distributions are attained seems to encourage users to experiment with different mappings — it becomes fun, rather than tedious, to try the effects of other distributions. The most important contribution of VFD, however, may well be its didactic value. The Fortran D model, with its two levels of mappings, confuses many traditional programmers. The ability to see a Fortran D statement and its graphical representation side by side appears to be a valuable learning aid.

At present VFD is only a prototype tool; a full implementation of a programming environment for Fortran D would link VFD with the companion compiler. Nevertheless, the tool is usable as a stand-alone application, from which the generated Fortran D statements can be cut and pasted into program source code.

The prototype has a couple of other shortcomings as well. Many parallel programmers make use of the concept of communications topology; that is, logical patterns of communications such as a tree, ring, torus, or cube. Although the current implementation of VFD allows the user to view the contents of individual processors, it does not visually support the concept of arbitrary topologies. Another problem is that only arrays up to rank three are currently supported. Since Fortran D restricts distributions to at most two dimensions (others must be collapsed), the mechanism devised to support the third dimension could be extended fairly easily to higher dimensions. Future extensions are also needed to support Fortran D's indexed mappings. This is potentially very difficult, since it is possible for the user to define arbitrary mapping relationships. The most likely direction is to provide navigational techniques such as those employed in visualizing dynamic data structures (e.g., [24, 31, 3]).

Nevertheless, VFD demonstrates that visual techniques can be integrated in a straightforward way into a Fortran programming environment. The concept of representing arrays as graphical objects which can be manipulated directly by the user can be applied to other parallel languages as well. Data distribution idioms, such as Vienna Fortran [17], HPF [14], the Yale Extension to Fortran90 [6], and the tiling directives from Kendall Square Research's Parallel Fortran [16], would be particularly amenable to such techniques.

Even more promising is the notion of extending the VFD visualizations to debugging and performance tuning activities. One of the most difficult aspects of Fortran D programming is the fact that debugging must be carried out against the "expanded" code, where data distribution specifications have been replaced by explicit message passing calls or SIMD modifications. A runtime tool which could map those calls back to the graphical framework originally manipulated by the programmer could be significant in improving the acceptability and ultimate popularity of Fortran D and similar parallel languages.

References

[1] Ambler, A. L. and M. M. Burnett. Influence of Visual Technology on the Evolution of Language Environments. IEEE Computer 22 (10): 9-22 (1989).

[2] Brewer, O., J. Dongarra and D. Sorenson. Tools to Aid in the Analysis of Memory Access Patterns for Fortran Programs. Parallel Computing 9: 25-35 (1988/89).

[3] Brown, G. P., R. T. Caring, C. F. Herot, D. A. Kramlich and P. Souza. Program Visualization: Graphical Support for Software Development. IEEE Computer, 18 (8): 27-35 (1985).

[4] Brown, M. and R. Sedgewick. Techniques for Algorithm Animation. IEEE Software 2 (1): 28-39 (1985).

[5] Chapman, B., P. Mehrotra and Hans Zima. Programming in Vienna Fortran. Proc. Third Workshop on Compilers for Parallel Computers, pp. 121-160.

[6] Chang, S., T. Ichikawa and P. A. Ligomenides. Visual Languages. Plenum Press (1986).

[7] Chen, M. C. and J. Wu. Optimizing Fortran-90 Programs for Data Motion on Massively Parallel Systems. Tech. Report DCS-TR-882, Yale University (January 1992).

[8] Davis, T. A. and E. S. Davidson. Pairwise Reduction for the Direct, Parallel Solution of Sparse Unsymmetric Sets of Linear Equations. IEEE Transactions on Computers 37 (12): 1648-1654 (1988).

[9] Dongarra, J., O. Brewer, J. A. Kohl and S. Fineberg. A Tool to Aid in the Design, Implementation, and Understanding of Matrix Algorithms for Parallel Processors. Journal of Parallel and Distributed Computing, 9:185-202 (1990).

[10] Dyer, D.S. A Dataflow Toolkit for Visualization. IEEE Computer Graphics & Applications 10: 60-69 (1990).

[11] Fox, G., S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng and M. Wu. Fortran D Language Specification. Tech. Report TR90-141, Dept. Computer Science, Rice University (April 1991).

[12] Glenn, R. R., J. M. Conroy and D. V. Pryor. Instrumentation for Massively Parallel MIMD Applications. Proc. ACM/ONR Workshop on Parallel and Distributed Debugging, pp. 234-236 (1991).

[13] Heath, M. T. and J. A. Etheridge. Visualizing Performance of Parallel Programs. IEEE Software 6 (5): 29-39 (1991).

[14] Hiranandani, S., K. Kennedy and C. Tseng. Compiling Fortran D. Communications of the ACM 35 (8): 66-80 (1992).

[15] High Performance Fortran Forum. Draft of High Performance Fortran Language Specification (November 1992). [Document available via anonymous ftp from Rice University; contact chk@cs.rice.edu.]

[16] Hough, A. A. and J. E. Cuny. Initial Experiences with a Pattern-Oriented Parallel Debugger. Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging. Published in ACM SIGPLAN Notices 24 (1): 195-205 (1989).

[17] Kendall Square Research. KSR Fortran Programming. Kendall Square Research Corporation (1992).

[18] Kimelman, D. N. and T. A. Ngo. The RP3 Program Visualization Environment. IBM Journal of Research and Development 35 (5/6): 635-651 (1991).

[19] Kondapaneni, P. K. A Visual Tool for Fortran D Programming. M.S. Thesis, Dept. Computer Science and Engineering, Auburn University (August 1992).

[20] LeBlanc, T. J., J. M. Mellor-Crummey and R. J. Fowler. Analyzing Parallel Program Executions using Multiple Views. Journal of Parallel and Distributed Computing 9: 203-217 (1990).

[21] Malony, A. D. and D. A. reed. Visualizing Parallel Computer System Performance. Tech. Report UIU-CDCS-R-88-1465, Dept. Computer Science, University of Illinois at Urbana-Champaign (September 1988).

[22] Myers, B. A. The State of the Art in Visual Programming and Program Visualization. Tech. Report CMU-CS-88-114, Dept. Computer Science, Carnegie Mellon University (February 1988).

[23] Open Software Foundation. OSF/Motif Style Guide. Prentice-Hall (1991).

[24] Pazel, D. P. DS-Viewer — An Interactive Graphical Data Structure Presentation Facility. IBM Systems Journal 28 (2): 307-323 (1989).

[25] Pancake, C. M. Graphical Support for Parallel Debugging. To appear in Software For Parallel Computation, ed. J. Kowali. Springer-Verlag (1993).

[26] Pancake, C. M. and S. Utter. Debugger Visualizations for Shared-Memory Multiprocessors. In High Performance Computing II, ed. M. Durand and F. El Dabaghi, pp. 145-158. Elsevier Science (1991).

[27] Pancake, C. M. and D. Bergmark. Do Parallel Languages Respond to the Needs of Scientific Programmers? IEEE Computer, 23 (12): 13-23 (1990).

[28] Pancake, C. M. and S. Utter. A Bibliography of Parallel Debuggers — 1990 Edition. ACM SIGPLAN Notices 26 (1): 21-37 (1991). [The bibliographic database is available via anonymous ftp; for information, contact pancake@cs.orst.edu.]

[29] Raeder, G. A Survey of Current Graphical Programming Techniques. IEEE Computer 18 (8): 11-25 (1985).

[30] Rohr, G. Using Visual Concepts. In [5], pp. 325-348.

[31] Shimomura, T. and S. Isoda. Linked-list Visualization for Debugging. IEEE Software 6: 44-51 (1991).

[32] Socha, D. M. L. Bailey and D. Notkin. Voyeur: Graphical Views of Parallel Programs. Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging. Published in ACM SIGPLAN Notices 24 (1): 206-215 (1989).

[33] Tuchman, A. and M. Berry. Matrix Visualization in the Design of Numerical Algorithms. ORSA Journal on Computing, 2 (1): 84-92 (1990).

[34] Tufte, E. R. The Visual Display of Quantitative Information. Graphics Press (1983).

[35] Tufte, E. R. Envisioning Information. Graphics Press (1990).

[36] Upson, C. *et al.* The Application Visualization System: A Computational Environment for Scientific Visualization. IEEE Computer Graphics & Applications 9: 30-42 (1989).