Data Distribution Visualization (DDV) for Performance Evaluation

Steven T. Hackstadt
Allen D. Malony

CIS-TR-93-21
October 1993

Department of Computer and Information Science
University of Oregon

Data Distribution Visualization (DDV) for Performance Evaluation

Steven T. Hackstadt
Allen D. Malony
Dept of Computer & Information Science
University of Oregon
Eugene, OR 97403
{hacks,malony}@cs.uoregon.edu

June, 1993
(Revised: October, 1993)

ABSTRACT

The next generation of language compilers for parallel architectures offers levels of abstraction above those currently available. Languages such as High Performance Fortan (HPF) and Parallel C++ (pC++) allow the programmer to specify how data structures are to be aligned relative to each other and then distributed across processors. Since a program's performance is often directly related to how its data is distributed, a means of evaluating data distributions and alignments is necessary. Since there is a natural tendency to explain data distributions by drawing pictures, graphical visualizations may be helpful in assessing the benefits and detriments of a given data decomposition. This paper formulates an experimental framework for exploring visualization techniques appropriate to evaluating data distributions. Visualizations are created using IBM's Data Explorer visualization software in conjunction with other software developed by the author. An informal assessment of the resulting visualizations and an explanation of how this research will be extended is also given.

1 Introduction

Modern parallel languages and compilers strive for at least three goals: ease of use, efficiency, and portability. Attainment of these goals has challenged the parallel computing community for some time. Most contemporary parallel programming languages and environments exhibit a trade-off between these goals. Those environments offering simplicity and efficiency are typically tied to a single parallel architecture, while parallel languages boasting portability can be more difficult to program. The next generation of parallel languages and compilers hopes to defeat the reasons for these trade-offs.

New languages such as High Performance Fortran (HPF) [4] and Parallel C++ (pC++) [1] offer cohesive data parallel programming abstractions, strive for efficiency, and have portability as as an underlying motivation in their design. Thus, they hope to attain all three goals identified above. While these languages offer levels of abstraction far removed from the physical architecture, they still give the programmer explicit control over such issues as data distributions, data alignments, abstract processor arrangements, and specification of parallel loop constructs. The results are languages whose compilers can capitalize on certain recent advances in compiler technology ([5], for example), but presently leave the critical task of distributing data across processors up to the programmer. Perhaps this signifies the parallel computing community's realization that automatic (and efficient) static data distribution is an extremely difficult task. There are those people who will undoubtedly applaud this step and consider user-specified data distributions a "feature." Equally likely, there are those who will dread being forced to figure out the best way to partition their data structures. Giving this ability (or burden) to the programmer, however, places the performance of thousands of parallel applications in potential jeopardy. Clearly, the distribution of data across processors is a deciding factor in the performance of the program operating on that data. Thus, programmers using new languages such as HPF and pC++ are going to demand some means of evaluating different data distributions.

The goal of this research is to demonstrate how performance visualization techniques can be developed to meet the needs of programmers requiring evaluations of data distributions. The work by Kondapaneni, Pancake, and Ward [8] demonstrates how data distributions and alignments in Fortran D (a language which contributed significantly to the design of HPF) may be specified using graphical representations of program data structures. They claim that the accuracy of data distributions specified by programmers using their visual programming tool were more accurate than those made by

programmers who did not use the tool. Furthermore, the graphical environment also encouraged experimentation with different mappings. These results strongly suggest that the *evaluation* of data distributions could benefit from graphical representations as well. To this end, we embark on the development of an experimental data distribution visualization (DDV) environment — a framework from which visualization techniques, evaluation criteria, and further research may arise.

The rest of this paper will proceed as follows. Section two will offer a brief overview of the DDV project and a summary of other relevant research. Section three will present the aspects of High Performance Fortran pertinent to this work. A discussion of the project's methodology in section four will be followed by an application in section five. Section six will offer some initial observations on the value of the visualizations which were generated, and a discussion of future work and the paper's conclusion will appear in section seven.

2 DDV Overview

2.1 Goals

HPF and pC++ offer data parallel extensions to Fortran and C++, respectively, allowing programmers to specify data distributions for the data structures of each language. In HPF the programmer distributes arrays only. In pC++, traditional C++ objects are the targets of distribution specifications. As a starting point, this research will focus on developing visualizations for HPF. High Performance Fortran offers a simple platform from which to begin this work since only arrays need to be considered. With that in mind, three primary goals can be established:

- *Develop criteria for evaluating data distributions in HPF.*
- *Keep visualizations meaningful to the programmer.*
- *Provide a research base from which more advanced topics may be pursued.*

For the first goal, metrics that will help the programmer determine the effectiveness of a given data distribution must be identified. Once this occurs, visualizations showing some or all of those metrics for a given distribution can be developed. In accordance with the second goal, these visualizations should demonstrate some consistency with the programmer's mental model of the data structure, the distribution, or other aspects of the problem. This is a critical goal which deserves further explanation.

One of the primary features of HPF is that it offers an abstraction far removed from any particular architecture. Thus, the specific architectural knowledge

required by the HPF programmer is far less than that required for other parallel environments. The consequence of this is that traditional performance visualizations based on physical processor load and low-level message passing, for example, may not be as meaningful to the HPF programmer. Thus, visualizations for HPF must demonstrate levels of abstraction that are consistent with the language's — and hopefully the programmer's — underlying conceptual model.

Although many features of HPF could potentially benefit from performance visualization, this research project is focusing on the visualizations for evaluating data distributions. As will be seen later, this paper investigates a problem of limited scope. It was mentioned above that HPF visualizations could be more effective if, to a certain degree, they were *dependent* on HPF's underlying programming and operational semantics. This suggests two additional areas of research. First, the notion of *source language-based visualizations* becomes evident. In other words, is it possible that an HPF program with directives for data distributions contains enough information to generate visualizations which are useful to the programmer? Second, if source language-based visualization is possible, can the generation of such visualizations be automated? Can a *visualization compiler* for HPF be built? These are research topics which may eventually benefit from the work being done here.

2.2 Approach

The work proceeded in five primary phases which are essentially echoed by the sections of this report. Initially, a review of HPF was conducted to determine syntax, distribution capabilities, programming model, and abstractions. A review of current visualization technology followed. Based on these two areas, a DDV visualization framework was established. This primarily consisted of carving the problem down to a manageable size for the limited time frame of this project. From here, an experimental visualization environment evolved, and various visualizations were developed using *Data Explorer*, a commercial data visualization software package by IBM. Finally, an informal assessment of the current work took place.

2.3 Related Research

Performance visualization has become a hotbed of computer science research. Tools such as ParaGraph [3], Pablo [9], and Sieve [10] have become popular because of their portability, scalable displays, and complete visualization environments. The displays offered by these products can be very effective for *general* parallel performance evaluation. Typically, however, the

visualizations are independent of the source-language used and do not, in any way, reflect the underlying conceptual model that may be inherent in an application, a programming model, or data structures.

As mentioned earlier, the visual programming tool for Fortran D developed by Kondapaneni, Pancake, and Ward [8] offers visual representations of data structures to aid in the specification of data distributions and alignments. They also demonstrate how source code can be processed to assist in building the graphical representations of data structures. While their work does not extend into the arena of performance evaluation, the effectiveness of graphically representing data distributions and alignments has immediate relevance and adds validity to this research.

The Program Visualization (PV) environment is a prototype tool developed by Kimelman and Sang'udi [7] at IBM's T. J. Watson Research Center. PV pursues the topic of integrating the generation of visualizations with the language compilation process. They propose a language-level program structure as a basis for more effective displays. Their environment parallelizes sequential Fortran code and generates traditional performance visualizations such as static call graphs, control flow graphs, and dependence graphs for subprograms. Since their version of Fortran does not allow user-specified data distributions, visualizations addressing this issue are not immediately relevant.

The research most pertinent to DDV is being done in the computer science department at Indiana University by Srinivas and Gannon [11]. They propose an interactive environment that aims to address many of the same questions about data distributions that this paper does. They also recognize the importance of providing visualizations in terms of the semantics of the application for new languages like HPF and pC++. Source code analysis and visualization of data structures are also components of this environment.

The work which is the focus of this paper integrates many of the concepts just presented. Currently, it is most similar to the work of Srinivas and Gannon [11], but in the future it will potentially integrate the research done by Kimelman and Sang'udi [7] as well. Kondapaneni *et al.* [8] offer support for the benefits of graphically representing data distributions, while the host of other visualization tools all emphasize the important role visualization now plays in performance evaluation. We are attempting to bring these two research areas together.

3 High Performance Fortran

3.1 Overview

High Performance Fortran is a parallel Fortran language that hopes to offer support for high performance on a wide variety of parallel processing architectures, including massively parallel SIMD machines (*e.g.*, Maspar's MP-1 or Thinking Machine's CM-2), distributed- and shared-memory MIMD architectures (*e.g.*, the Intel Paragon and the Cray Y-MP C90), vector processors, and other architectures. Version 1.0 of the language specification [4] was released on May 3, 1993, by the HPF Forum, a coalition of industrial and academic groups representing most commercial vendors, several government labs, and many university research groups.

HPF is a set of extensions and modifications to Fortran 90 [6] that will offer data parallelism in the form of explicit parallel loop constructs, code tuning for various parallel architectures through a set of extrinsic procedures, and high performance on MIMD and SIMD machines through data distribution and alignment capabilities which the user spceifies with compiler directives.

3.2 The Programming Model

While there are many features of HPF which make it an interesting topic of discussion, this work is concerned only with its data distribution features. It is well known that modern parallel architectures achieve their best performance when data accesses exhibit high locality of reference. A programmer should, therefore, try to limit the frequency with which a processor must obtain data from other processors. Data distributions hold the key to achieving this goal. HPF allows the programmer to specify aspects of the distribution process. The primary directives and their functions are given in Figure 3.1.

HPF Directive	Function
`PROCESSORS`	Declare an n–dimensional mesh of abstract processors
`ALIGN`	Specify relationships between multiple arrays
`DISTRIBUTE`	Specify how arrays (or groups of arrays) are distributed across processor sets.

Figure 3.1 — Primary HPF Directives

The PROCESSORS directive allows the user to declare an abstract set of processors onto which distributed arrays will be mapped. The abstract processor set takes the form of an n-dimensional mesh with no inherent interconnection network. Next, relationships between different data structures are specified with the ALIGN directive. An ALIGNment offers a common structure to which several arrays are oriented. Finally, using the DISTRIBUTE directive, ALIGNments (*i.e.*, groups of related arrays) are mapped onto the abstract set of processors. Thus, the HPF programming model can be illustrated as in Figure 3.2 [4].

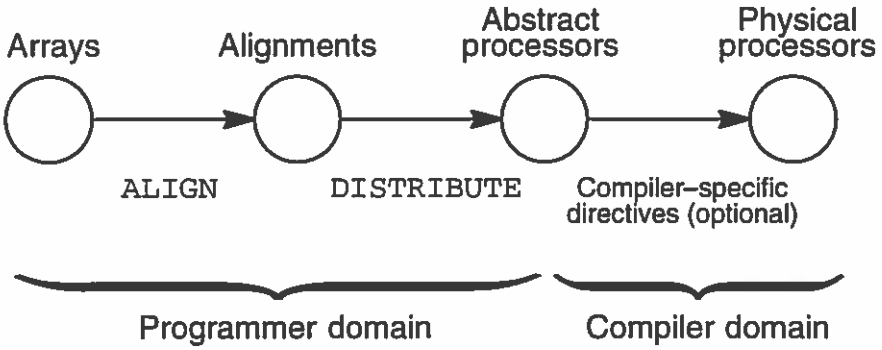


Figure 3.2 — HPF Model

From the programmer's perspective, this is a two-level mapping: first, data structures are aligned with one another, and then these groups are distributed onto a set of abstract processors. The compiler takes responsibility for mapping the abstract processor set onto the physical processors. (The HPF Forum suggests that commercial HPF compilers may wish to offer compiler-specific directives so that the user has some say in how abstract processors are mapped to physical processors.) From this, the underlying programming model becomes evident. Programs are written as if running on an arbitrarily sized, n-dimensional mesh of processors. This is consistent with the ideal data parallel programming paradigm where the amount and type of data determines the processing resources needed.

As a side note, it is important to realize that specifications for processors, distributions, and alignments in HPF are expressed as compiler *directives*. Thus, the programmer is advising the compiler on how to partition data structures; a compiler is not obligated to follow such directives.

The HPF model is dependent on two assumptions that are reasonable for current architectures. First, an operation involving multiple data elements will be faster if the elements are all on the same processor. Second, many such

operations can be done simultaneously if they can be performed on different processors.

3.3 Examples of HPF Directives

This section will offer a few examples of HPF directive syntax for specifying processors and distributions. Alignments will not be discussed here as they are not currently incorporated into the visualization model. Visualizations which would incorporate the concept of alignments stand to be considerably more complex than the displays discussed here.

3.3.1 Processors

The PROCESSORS directive is used to declare an n-dimensional mesh of abstract processors. The following are examples of some possible processor arrangements.

```
!HPF$ PROCESSORS A_SCALAR
!HPF$ PROCESSORS A_VECTOR(16)
!HPF$ PROCESSORS A_GRID(32,32)
!HPF$ PROCESSORS A_CUBE(8,8,8)
!HPF$ PROCESSORS A_MESS(2,5,3,4,8,1)
```

The A_SCALAR processor specification declares a scalar processor (which might be mapped to a parallel machine's front-end processor). A_VECTOR corresponds to a linear vector of processors. As can be seen from the last example, processor arrangements do not have to be uniform in all dimensions.

3.3.2 Distributions

The DISTRIBUTE directive allows the programmer to specify how arrays or groups of arrays are distributed. The following examples will specify distributions independent of any particular set of processors. The next section (3.3.3) will combine the the PROCESSORS and DISTRIBUTE directives.

```
      INTEGER CHESS(8,8)
      INTEGER GO(19,19)
!HPF$ DISTRIBUTE CHESS(BLOCK,BLOCK)
!HPF$ DISTRIBUTE GO(CYCLIC,*)
```

These two examples distribute the arrays CHESS and GO in significantly different ways. The CHESS array will be chopped into blocks whose size will be determined by the compiler and then distributed in both dimensions. The GO array, however, will only be distributed across the first dimension. That is, no distribution across the columns of this array will occur; entire rows will be

spread across processors in a cyclic fashion. In the first case, the blocking factor has been left unspecified so that the compiler will try to determine the best value to use. By not specifying a parameter for the CYCLIC distribution, the factor defaults to one. The next section offers an example of a cyclic distribution factor.

3.3.3 Combined

The DISTRIBUTE directive can be used to distribute an array across a specific set of abstract processors. The following example demonstrates this capability.

```
        INTEGER CARDS(52)
!HPF$ PROCESSORS PLAYERS(4)
!HPF$ DISTRIBUTE CARDS(CYCLIC(3)) ONTO PLAYERS
```

The net effect of these directives is to "deal" three cards (that is, elements of the array CARDS) to each player (or, processor in the processor set PLAYERS).

For more information or additional examples of HPF directives, the reader is referred to the HPF Language Specification [4].

3.4 Computations in HPF

The final aspect of HPF which must be addressed is the manner in which computations are carried out. Bozkus *et al.* [2] propose four possible models of computation based on the locations of left-hand and right-hand data elements. Given the generalized form of an assignment statement, *A: lhs = rhs*, Figure 3.2 illustrates the four possible cases.

P_i denotes processor i. The processor containing "lhs" owns the data element on the left-hand side of the assignment; the processor(s) with "rhs" own data elements which are required by the computation; the processor containing the "=" is the processor which performs the computation. Annotated arrows indicate the interprocessor communication required with respect to when the computation is performed (before or after).

Given a data distribution, the processor responsible for performing a computation is not specified by HPF; that is, it is up to the compiler to determine which processor will perform a given computation. As noted in Figure 3.2, one of the more popular parallel computation models is known as *owner computes*. Under this scheme, the processor which "owns" the left-hand side data element (*i.e.*, the data element to which the assignment is being made) is the one that will perform the computation. Thus, before the computation can occur, all processors holding data elements which occur on

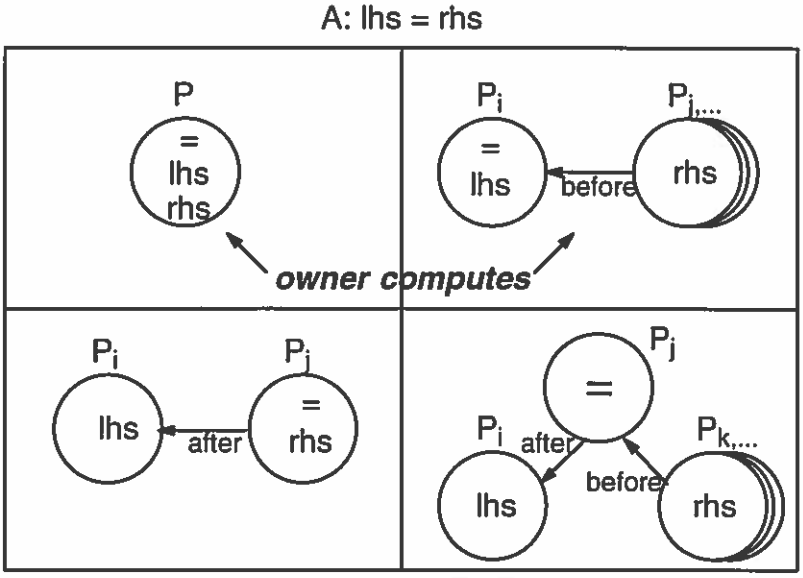


Figure 3.2 — Parallel Computation Models

the right-hand side of the assignment must send their data to the owner processor. A degenerate case of owner computes occurs when *all* data elements are locally owned. Thus, the top half of Figure 3.2 illustrates cases of the owner computes rule.

Since the manner in which assignments and computations are carried out is compiler-dependent, this work will generally assume that the owner computes rule is used. This assumption is evident only in the sample application described later.

4 DDV Methodology

The methodology of the DDV project consisted of two primary tasks which will be discussed in the following sections. First, a limited DDV problem domain was established. Then, an experimental environment where data distribution visualizations could be built, refined, and evaluated was built. Figure 4.1 diagrams the process that was followed to define the DDV problem.

4.1 Establishing the DDV Problem Domain

Establishing a problem domain for data distribution visualization consists of deciding which HPF features will be addressed by the visualizations, how data distributions are to be evaluated, and what properties the visualizations should have. The following sections will identify and justify the decisions which were made.

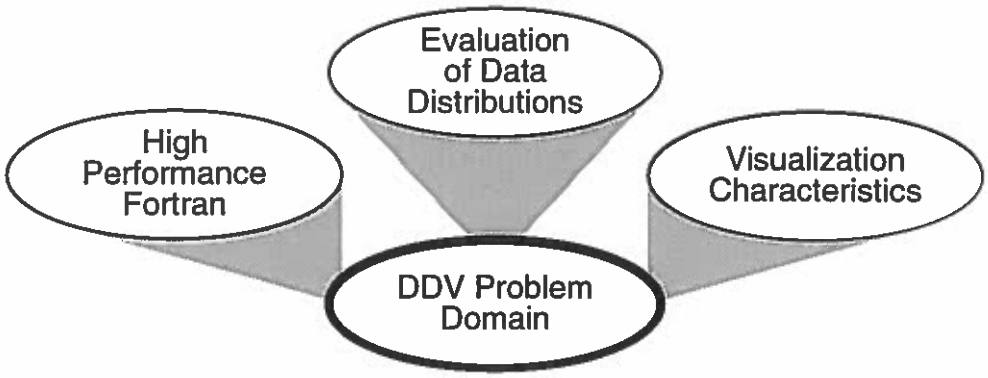


Figure 4.1 — DDV Problem Domain

4.1.1 High Performance Fortran

For this initial attempt, it is important that the problem be accessible. For this reason, visualizations are limited to two-dimensional data arrays and two-dimensional sets of processors. This results in simple, obvious mappings from data structures and processor sets to graphical objects. (Similarly, an upper-bound of three dimensions was set in [8].)

In addition, data structure alignments are not considered. The current visualization system considers only individual arrays. Thus, relationships to other arrays, specified with the `ALIGN` directive, are currently irrelevant, though such relationships could undoubtedly play a role in future performance visualizations.

In HPF, programmers can dynamically redistribute arrays in the middle of their program. To maintain simplicity, DDV considers static distributions only.

Finally, as mentioned earlier, the owner computes rule will be assumed as the computational model for sample applications (which will be discussed later).

4.1.2 Evaluating Data Distributions

The evaluation of data distributions is essentially a load balancing problem. The programmer wants to make sure that data structures are distributed in such a way that no single processor is excessively burdened with data requests. In accordance with the assumptions underlying HPF, the programmer also wants to maximize local data accesses and minimize the number of remote data operations that a given processor must perform.

To evaluate a data distribution for a given application, the programmer needs to formulate answers to two questions:

- *What memory reference patterns are exhibited by the application?*
- *Given the data distribution, how do these reference patterns manifest themselves on the set of processors?*

Thus, it stands to reason that the type of accesses being made to data structures (*i.e.*, reads or writes) and the type of communication associated with that operation (*i.e.*, local or remote) will play an important role in determining whether a given distribution is good or bad. If visualizations can be devised that show reference patterns for both the data structure and the processors across which the data is distributed, then the HPF programmer can answer the questions above. To this end, DDV focuses on collecting information about data reference patterns, and then determines which processor has that data element by the data distribution which was specified.

4.1.3 Visualization Characteristics

One of the primary goals set out earlier in this paper was that visualizations should incorporate the programmer's mental model. This applies to both data structures and processors. The limitations pertaining to HPF set out above make this task much easier. By placing an upper-bound of two dimensions on arrays and processors, visualizations should be able to include easily the traditional representation of arrays. The creators of the visual programming tool for Fortran D [8] point out the importance that should be placed on maintaining consistency between graphical portrayals of arrays and the programmer's mental model of arrays. They also indicate that a grid of cells arranged in rows and columns is the most obvious and intuitive representation of such structures.

In order to answer fully the questions posed above, visualizations should incorporate the passage of time. In general, this means that there should be some component of animation in the visualizations.

Finally, the ability to show multiple dimensions of the problem in a single visualization may be useful in some circumstances. At the very least, this should be an option for the programmer.

Given the establishment of the DDV problem domain, the problem at hand is well-defined and approachable. We now turn to a discussion of the experimental environment which was developed.

4.2 Building an Experimental DDV Environment

In the following sections, a detailed examination of how the DDV environment currently operates will be presented. Figure 4.2 offers an overview of the steps from application to visualization.

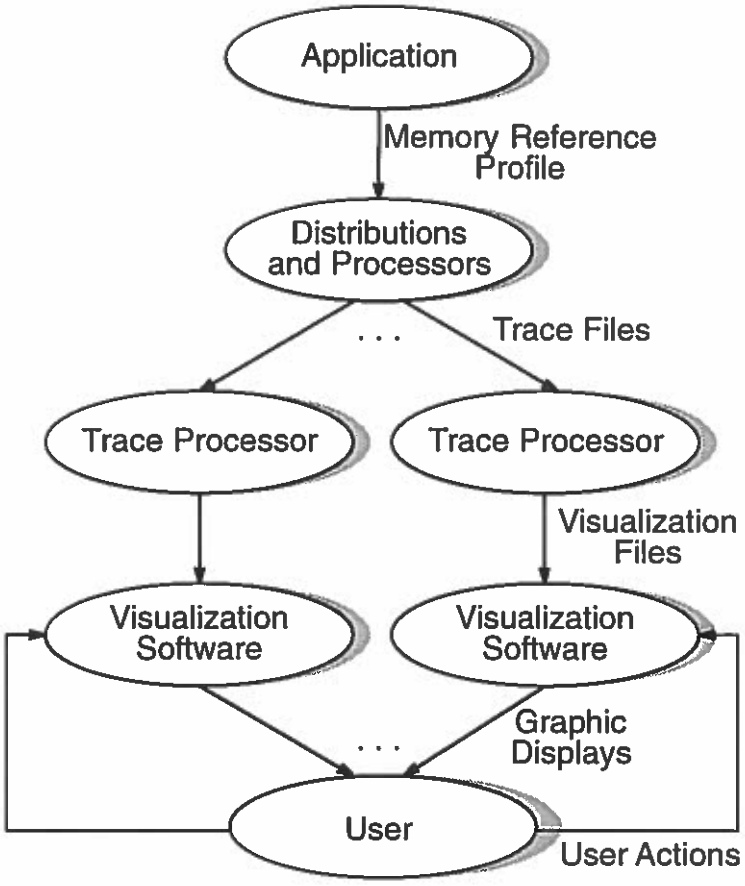


Figure 4.2 —DDV Processing

4.2.1 Creating Memory Reference Profiles

As with the visual programming tool for Fortran D [8], DDV encourages the programmer to experiment with different distributions for a given application. As shown in Figure 4.2, the first step in this process is to create a memory reference profile for an application. The current environment does not actually handle parallel programs; rather, a sequential application is used and the owner computes rule is imitated by calls to a `TRACE` function (see below). At desired points in the code, the programmer inserts function calls to generate profiling information about the particular memory operation(s) being performed and the data elements being referenced. To mimic owner computes, the profiling information is made relative to some other data element — in particular, the element appearing on the right hand side of the assignment. This process could be accomplished automatically with current program transformation technology.

While the profile may not actually represent the parallel execution, it identifies the reference patterns that the application exhibits. Better yet, the

reference profile is independent of any particular data distribution. Thus, a single profile can lead to the creation of any number of trace files which *are* distribution-specific. If a real parallel application was to be used, then the TRACE function could be easily modified to generate distribution-specific information, and the memory reference profile would be unnecessary.

The TRACE function accepts a time stamp, the location of the "owner" element, a range of referenced data elements, and the operation (read or write) performed. (Note that whether the operation required local or remote communication is unknown until a distribution is specified in later steps.) Figure 4.3 shows the function prototype and an example invocation.

Trace function prototype

```
void TRACE(   int time,
              int ownerRow, int ownerCol,
              int minRow, int maxRow,
              int minCol, int maxCol,
              int op)
```

Trace function call

```
TRACE(Time, j,j, j+1,row-1, j,j, 0);
```

Figure 4.3 — DDV Trace function

4.2.2 Building Trace Files

Once a memory reference profile has been established, any number of trace files can be generated. Each trace file is specific to a single data distribution and processor arrangement. Thus, the process of building a trace file requires a program that understands HPF's data distribution functions. The HPF language specification [4] offers formulaic definitions of the distribution options discussed earlier in this paper. These formulas provide the mapping from data elements to processors in a single dimension.

First, define *c(j,k)*, the ceiling division function, by

$$c(j,k) = \left\lfloor \frac{j + k - 1}{k} \right\rfloor .$$

Let d represent the size of the data array in a certain dimension, r, and let p be the size of the corresponding dimension in the processor array. (Assume all dimensions have a lower bound of 1.) Then, the specification BLOCK(*m*) means that data element j of dimension r will be mapped to the abstract processor at location *c(j,m)*, provided that $mp \geq d$. Similarly, CYCLIC(*m*)

means that the data element in location j of dimension r will be owned by processor $1+((c(j,m)-1) \bmod p)$.

Figure 4.4 gives a small sample from a trace file. The trace file allows for multiple operations in a given time step. In this fashion, the annotation of a sequential application can be used to mimic the operation of the parallel application.

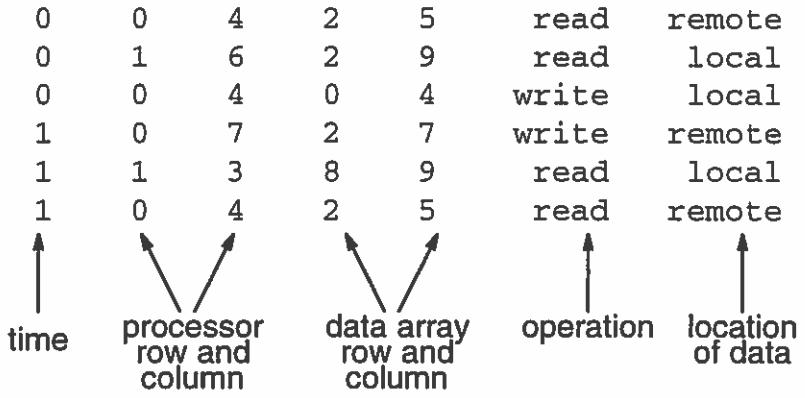


Figure 4.4 — Trace file fragment

The first record of the trace fragment in Figure 4.4 indicates that data element (2,5) was referenced. The data element is owned by processor (0,4), and it was read by a remote processor. This record indicates that processor (0,4) had to service a remote access, not that it initiated one. The processor performing the operation is not currently available, nor is it relevant to the problem at hand. (It would be a simple extension to add that information, though.) The reason for this is that the processor making the request is dependent on the computational model being used in that instance. We are not so much interested in determining the data *needed* by individual processors as we are interested in the types of data *accesses* being made for each element in the array. The latter information is what will help the programmer determine where data elements *should* reside in the distribution.

4.2.3 Creating Visualization Files

The next step to data distribution visualizations is to transform the trace file into a visualization file. Essentially, this step creates a data file that IBM's Data Explorer software can process. From the information in the trace files discussed above, this step keeps track of running totals and averages of local reads, remote reads, local writes, and remote writes for each element in the data array and the processor set. This information is present for each time step in the trace file. Thus, the totals and averages are computed over the time

that has expired up to that point in the trace. In this way, an animation of the application's memory references may be constructed.

4.2.4 Building a Visualization Framework

The final step in the visualization process is to import the visualization file into Data Explorer (DX) with a DX program. Data Explorer provides its own programming environment in which visualization applications can be built. Figure 4.5 is an image of the main control panel from the DDV visualization application. The user has control over numerous aspects of the visualization.

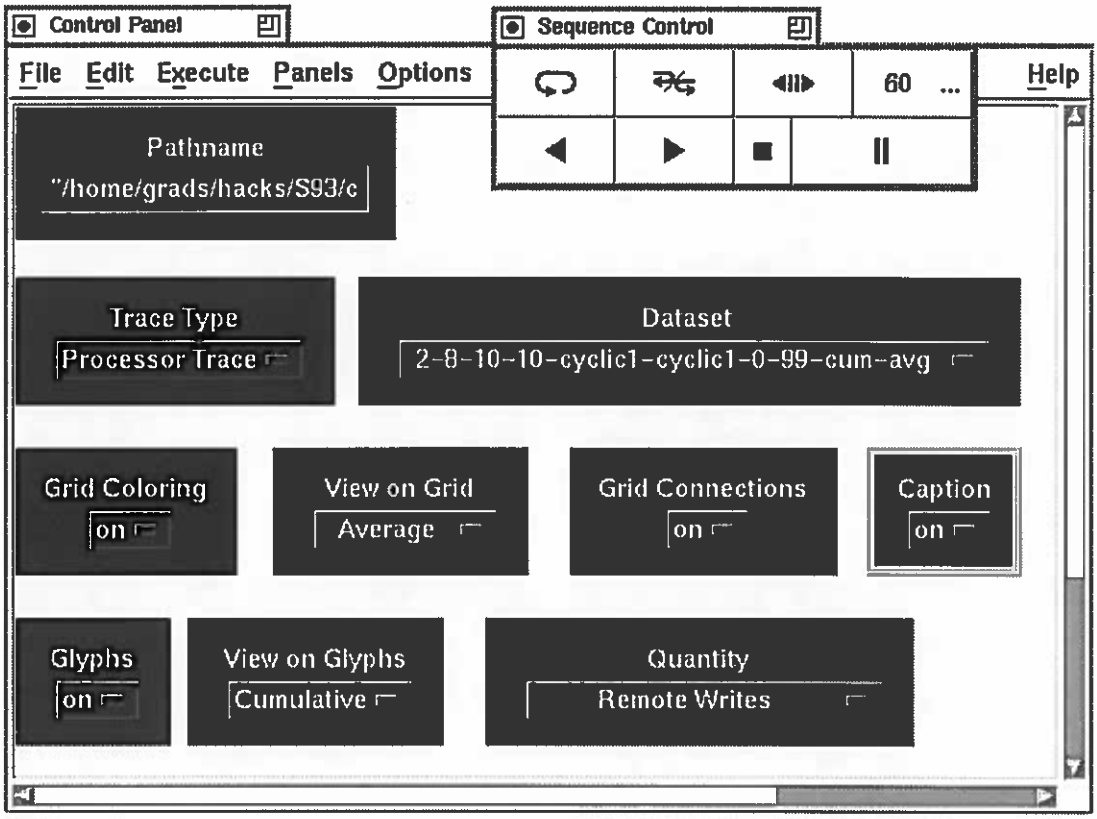


Figure 4.5 — DDV Control Panel

The visualizations currently display both statistics (cumulative and average) of a given trace quantity. One statistic is mapped to a colored background on the grid, and the other is mapped to "glyphs" which float in front of the grid. (The graphic displays will be discussed in more detail later.) The user can control several aspects of the display. Some of these are listed here.

- *The trace type to view (data or processor)*
- *The dataset to view (the user can easily add additional traces to the control panel)*

- *The quantity to be displayed display (e.g., local reads, all remote accesses, all reads and writes, etc.)*
- *The statistics to be displayed (cumulative and/or average)*

The user can also control the animation with the "Sequence Control" box. The controls are simple and intuitive, similar to those used in audio cassette and compact disc players.

In the actual visualizations, quantities are represented by the size and color of the glyphs, and by the color of the background grid coloring. (Note that glyphs currently contain redundant information. That is, the size and color of the glyph represent the same quantity.)

4.2.5 Notes on the Displays

The best way to understand the actual displays that are generated is to present a sample application as done in the next section. Unfortunately, the displays contained herein are limited to greyscale, which poorly reflects the actual color content of the images. For this reason, a set of full-color GIF (Graphics Image Format) files is available from the authors. (Contact hacks@cs.uoregon.edu.) Another dimension of these visualizations that can not be gleaned from either the greyscale images here or the full-color snapshots is the animation which takes place through the life of the visualization.

5 Application — Gaussian Elimination

To demonstrate the effectiveness of these visualizations, a sample application will be presented here. As was mentioned earlier, the current DDV environment is set up to operate with sequential programs. By carefully annotating this code, a pseudo-parallel trace file can be generated.

5.1 The Data Distribution

In the following, visualizations built from a Gaussian elimination algorithm will be presented. The algorithm operates on a 10 x 10 data array which is distributed on a 2 x 8 grid of processors in a cyclic fashion (for both rows and columns). The quantity being displayed in the images consists of all memory accesses (that is, local and remote, reads and writes).

Let us consider the distribution of data which results from the distribution and processor arrangement given above. Figure 5.1 specifies the processor which owns each element of the 10 x 10 array by assigning each processor a

unique number. It is important to note that processors 0, 1, 8, and 9 own many more elements than the other processors because columns are distributed in a cyclic manner.

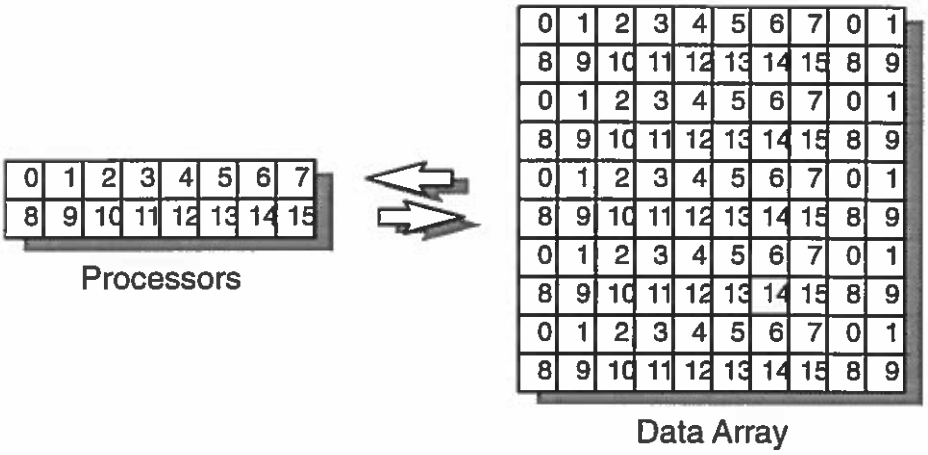


Figure 5.1 — Data distribution for Gaussian elimination

5.2 The Algorithm

The basic idea behind Gaussian elimination is to apply elementary row operations (scalar multiplies and row additions) to transform an invertible matrix into the identity. (Practically speaking, this is used when the original matrix is augmented with the identity matrix. If the same operations are applied to both matrices, the inverse of the original matrix will result in the augmentation.) The pseudo-code for Gaussian elimination appears in Figure 5.2.

```
for each column j from 0 to MAX, do
   make diagonal element (j,j) equal to 1
   for each row i from j+1 to MAX, do
      make element (i,j) equal to 0

for each column j from MAX to 0
   for each row i from j-1 to 0, do
      make element (i,j) equal to 0
```

Figure 5.2 — Gaussian elimination pseudo–code

5.3 Source-code Annotation

A fragment of code from the Gaussian elimination algorithm used to generate the visualization presented in the following sections appears in Figure 5.3.

The piece of code illustrates the transformation that takes place during the annotation process. The loop is one part of the Gaussian elimination

```
1   c = 1.0/B[j][j];
2   for (col=0; col<MAX; col++) {
3      /* scalar mult. the elements of row */
4      B[j][col] = c*B[j][col];
5
6      /* each owner read a copy of B[j,j] */
7      TRACE(Time, j,col, j,j, j,j, 0);
8      /* read own copy of B[j,col] */
9      TRACE(Time, j,col, j,j, col,col, 0);
10     /* wrote B[j,col] */
11     TRACE(Time, j,col, j,j, col,col, 1);
12
13     /* do the same to the augmentation */
14     M[j][col] = c*M[j][col];
15  }
16  Time++;
```

Figure 5.3 — Annotated version of one Gaussian elimination loop

algorithm. In this example, each row is being scaled by the the reciprocal of the element on the main diagonal. While the sequential algorithm makes an assignment to the variable `c`, the calls to `TRACE` mimic what might actually happen in a parallel version of the algorithm by having each of the "owners" — referenced by the data element at (j,col) — request copies of that value. Note also that the `Time` variable is not incremented until after the loop, suggesting that the operations in the `for`-loop could be done simultaneously in a parallel implementation.

5.4 Generating the Displays

The annotations are compiled along with the source-code, and a file containing a stream of memory references augmented with some additional information results after the application is run. This memory reference profile is independent of any data distribution, and it represents the pattern of memory accesses that this implementation of Gaussian elimination displays under a loose interpretation of the owner computes rule. The profile is used to create several different traces — one for each distribution that the user wants to evaluate. The trace files are transformed into visualization files which are imported into Data Explorer where the user can control many aspects of the resulting animation.

5.5 The Visualizations

At this point, we are finally prepared to present a sequence of actual visualizations which resulted from the Gaussian elimination program. (The filenames given after references to figures containing greyscale displays correspond to files in the directory discussed in Section 4.2.5.)

Figure 5.4 (`data22.gif`) illustrates the data array early in the execution of the Gaussian elimination algorithm (time=22). The background (when seen in color) represents the average number of memory references, while the glyphs (size and color) keep track of the cumulative number of references. With element (0,0) in the upper left corner the access patterns of this phase of the algorithm are already evident. As the algorithm proceeds, the glyphs in each column below the main diagonal grow larger (and change color) as the column elements below the diagonal are zeroed-out, and the row elements are adjusted accordingly.

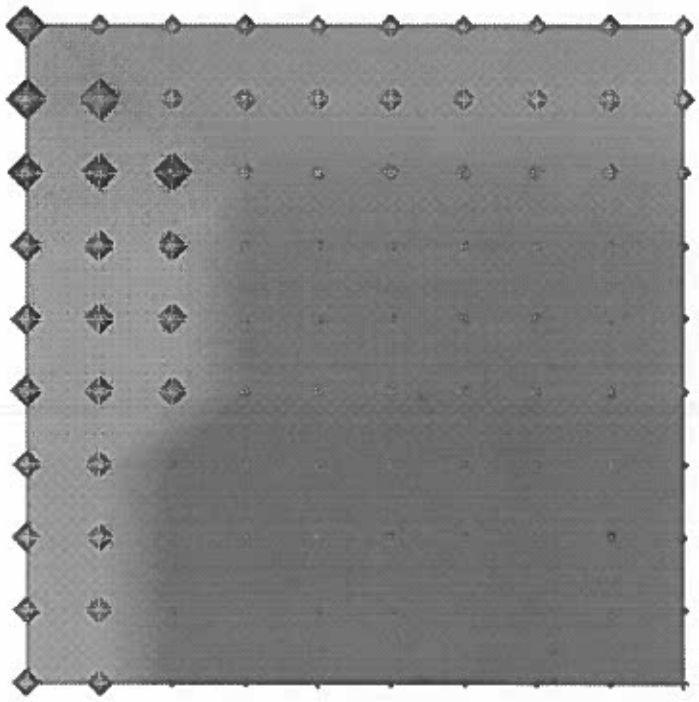
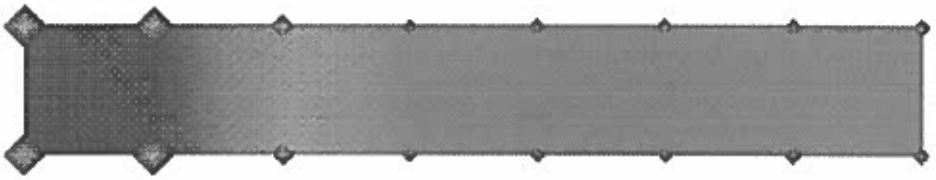


Figure 5.4 — data22.gif

Figure 5.5 (`proc22.gif`) displays the state of the processors at the same time as Figure 5.3. It can be seen that the size of the glyphs on the left side of the

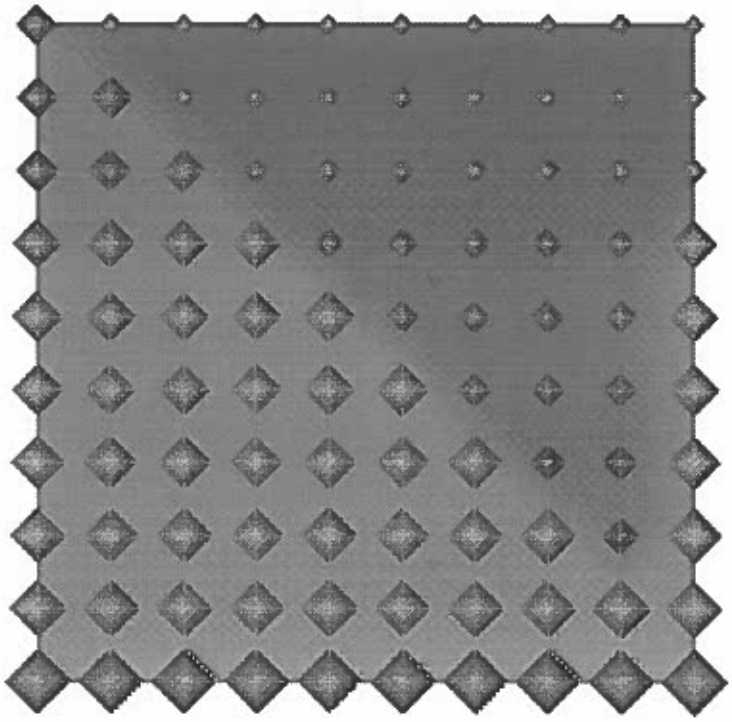
grid are larger than the others. At this point, it is too early to tell if this was due to a poor distribution or the particular phase of the algorithm.



Processor Trace: All Reads and Writes, time=22

Figure 5.5 — proc22.gif

Figure 5.6 (`data60.gif`) shows the data array at time 60 of the algorithm. The first phase of the Gaussian elimination has been completed and the second phase is just beginning. Thus, all of the elements on the main diagonal are now 1, and all elements below the diagonal are 0. The algorithm is proceeding to zero-out all matrix elements above the main diagonal by working its way back through the columns in reverse order, toward the origin of the array. Figure 5.7 (`data60rw.gif`) shows a different quantity being



Data Trace: All Reads and Writes, time=60

Figure 5.6 — data60.gif

displayed on the data array. Rather than showing *all* memory references, Figure 5.7 is showing *only* remote writes. The pattern of remote writes is considerably different than the overall memory reference pattern illustrated by Figure 5.6.

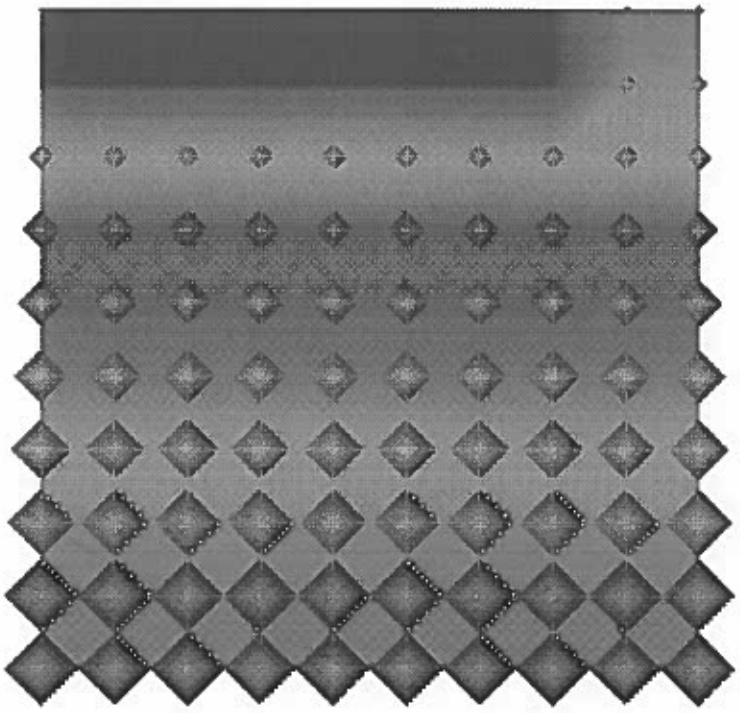


Figure 5.7 — data60rw.gif

Figure 5.8 (`proc60.gif`) again shows the status of the processor grid. The imbalance generated by the cyclic distribution is becoming more and more evident by this point in the animation. The other processors appear to be fairly well-balanced, though.

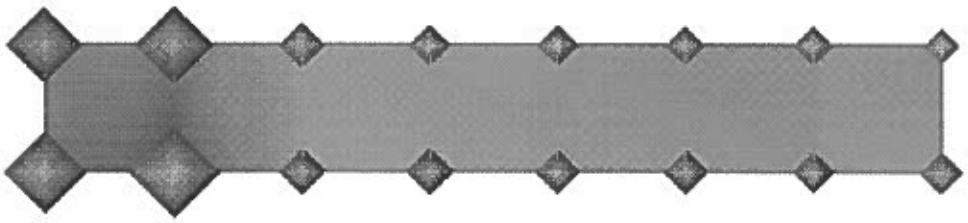


Figure 5.8 — proc60.gif

By time 89, as seen in Figure 5.9 (`data89.gif`), the algorithm is nearing completion with only two columns remaining to be processed. It is evident from this display that overall, the Gaussian elimination algorithm accesses the array in a fairly uniform manner. That is, there are no elements that receive excessive attention, in the long run. However, the earlier frames of the animation clearly show that the algorithm progressed in certain phases.

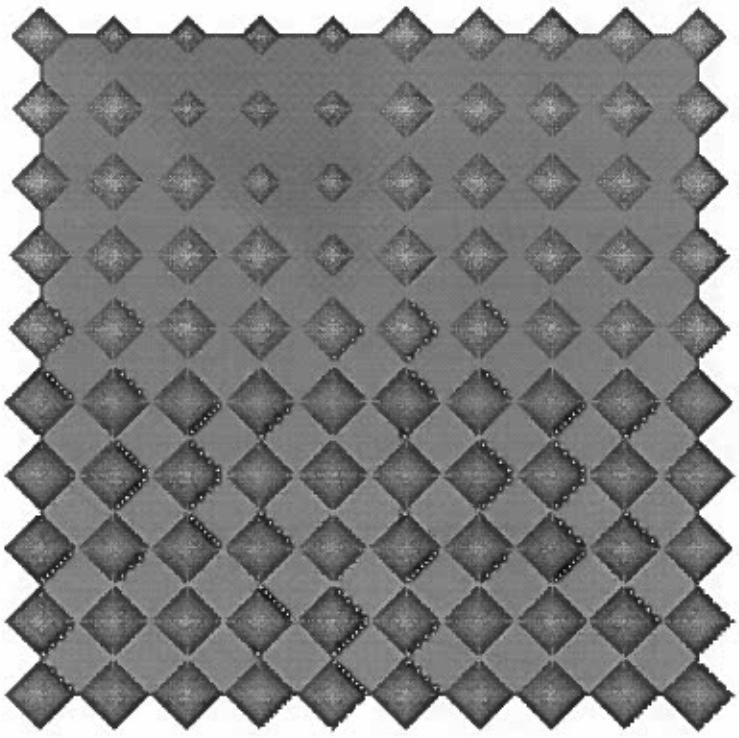


Figure 5.9 — data89.gif

Finally, Figure 5.10 (`proc89.gif`) illustrates the sorry state of the processor grid by the end of the algorithm. The load imbalance introduced by the cyclic distribution has clearly overtaxed the processors in the two left columns of the grid.

We have already come to some conclusions regarding the effectiveness of a cyclic distribution in this case. Using what has been learned and the DDV tools discussed earlier, the programmer could now go back and create a new trace file using a different distribution or a different processor arrangement. In this way, experimenting with different distributions is simplified and the

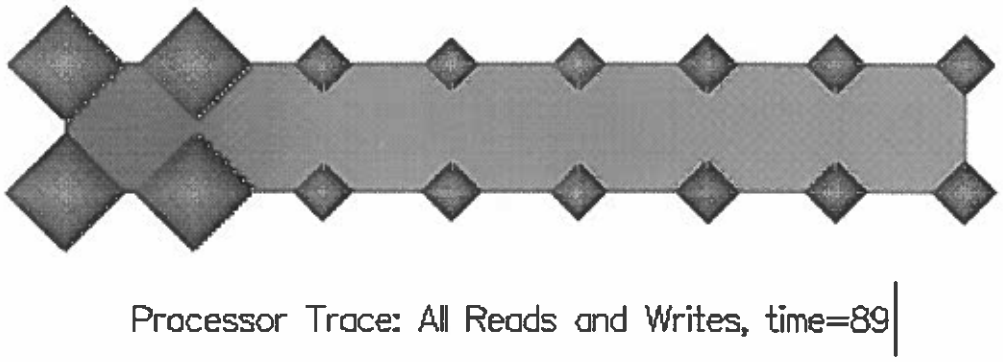


Figure 5.10 — proc89.gif

programmer undoubtedly gains insight not only to the algorithm, but to which distributions are most effective for it.

6 Observations

In this section, some informal observations on the effectiveness and utility of the DDV environment will be made. This will take place in three categories: usefulness, understandability, and scalability.

6.1 Usefulness

As was shown in the previous section, the visualizations generated from the DDV environment provided a good overview of memory reference patterns exhibited by the application. Depending on the quantity that the programmer is viewing, a sense of the interprocessor communication required is easily achieved. By offering the programmer visualizations of both the data structure and the processor set holding the data, one can effectively assess many characteristics of a given data distribution.

While reference patterns can be extracted directly from the animated displays, the size of messages being passed between processors remains obscured. Similarly, contention for data elements can not be seen. While we have not focused on these metrics in this inital work with HPF data distributions, they are still of interest to programmers.

6.2 Understandability

Since the problem domain limited the data structures and processor arrangements to two dimensions, these displays were able to take advantage of the obvious way to visualize such structures (*i.e.*, by using a grid). For this reason, the displays are fairly simple to understand. The images shown here

clearly lack labelling that would be present in a more complete environment, though. The actual DDV displays contain color bars for each quantity being measured, but they were omitted in this paper since color could not be shown.

It was mentioned earlier that the glyphs contain redundant information by having the size and color controlled by the same quantity. This could have both positive and negative effects. To novice users, it may reinforce the magnitude of the particular quantity. To more advanced users, it may be wasteful to have the same quantity represented twice in the same visualization.

6.3 Scalability

Finally, a very important topic in performance visualization is the scalability of the displays. Certain aspects of these visualizations are scalable while others are not. The glyphs, for example, do not offer a high degree of scalability. While very effective for small grids, the information is lost when grid size goes much above fifteen in one dimension. In particular, the variance in the size of the glyph becomes so minute that little or no information is conveyed. Thus, glyphs may be more appropriate for processor set since they tend to be smaller than data arrays.

The background grid coloring (not evident in the greyscale images included in this report) represents a more scalable visualization tool, however. By letting the color of the grid at each data element correspond to the quantity being measured, a global picture of memory references can be achieved for both small and large grids. In fact, the display actually seems to improve as the grid grows larger, up to a certain point. By letting the appropriate quantities color the grid background, the "hot" and "cool" spots of the data structure emerge. Also, Data Explorer offers zooming capabilities which allow the user to focus on particular regions of the data or processor structure.

7 Future Work and Conclusions

The work presented here can be extended in several ways, especially since a very limited problem domain was pursued by this research. Three areas of future work can be identified in terms of adding different visualizations, extending the model, and expanding the original problem. Each of these will be discussed below.

7.1 Additional Visualizations

The visualizations presented in this report had as a primary goal the maintenance of consistency with the programmer's conceptual model.

Additional visualizations may choose to deviate from this goal to reveal relationships that are not evident in a simple two-dimensional grid. It is still important that the programmer have access to displays like the ones presented here, but there could be much to gain from advanced displays which are not necessarily formulated from a programmer's conceptual view.

Currently, displays for data and processor structures are independent of one another. While driven by the same trace file, the displays can not be viewed simultaneously. It would be desirable to have a display that integrated the data and processor information. One way that this could be accomplished would be to take advantage of the redundancy currently present in the glyphs. For example, the glyph color could correspond to the processor which owns that data (hence, the visualization would also represent the data distribution itself), the grid background could be one quantity (processor or data), and the glyph color could be some other quantity (processor or data). This would be a fairly simple extension to the current DDV environment.

7.2 Extending the Model

Another potential source of future research would be to loosen some of the restrictions on HPF. For example, representation of three- and higher-dimensional arrays would be useful. The notion of data alignments could be incorporated into the visualization environment, allowing the relationships between multiple data structures on the same processor set to be visualized. Another extension would be to allow the dynamic redistribution and realignment of data during the execution of an application. In this case, at some point in the animation the data would be rearranged and an entirely new animation would ensue. Finally, including special visualizations for HPF's explicit parallel loop construct, `FORALL`, might be useful.

7.3 Expanding the Original Problem

The original problem motivating this research was the need for visualizations which would help programmers evaluate data distributions in new languages like HPF and pC++. Section 2 briefly mentioned two extensions to this topic. In particular, one branch of additional research could investigate further the degree to which source-code can drive the generation of visualizations, a process that the author has termed *source-based visualizations*. Another possible extension is an investigation into whether these types of visualizations can be created automatically by a *visualization compiler*. Some projects are already looking at these areas of research [7][11].

7.4 Conclusions

The research presented here has as its goal the development of an experimental environment that can help to assess the role that performance visualization can play in the evaluation of data distributions. We have informally demonstrated that performance visualization techniques can be developed that will benefit the programmer responsible for writing efficient HPF programs. The important role played by data distributions in determining the efficiency of parallel applications demands that tools which assist in evaluating distributions be developed. The Data Distribution Visualization, or DDV, environment developed here is a large step toward the development of such products.

8 References

[1] F. Bodin, P. Beckman, D. Gannon, S. Narayana, S. Yang. *Distributed pC++: Basic Ideas for an Object Parallel Language*. University of Rennes.

[2] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M. Wu. *Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers.* Northeast Parallel Architecture Center, Syracuse University, Technical Report SCCS–444, March, 1993.

[3] M. Heath and J. Etheridge. *Visualizing the Performance of Parallel Programs*. IEEE Software, September, 1991.

[4] High Performance Fortran Forum. *High Performance Fortran Language Specification*, Version 1.0. Rice University, May, 1993.

[5] S. Hiranandani, K. Kennedy, and C. Tseng. *Compiling Fortran D for MIMD Distributed–Memory Machine*. Communications of the ACM, Vol. 35, No. 8, August, 1992.

[6] ISO. *Fortran 90*. May, 1991. [ISO/IEC 1539: 1991 (E) and now ANSI X3.198–1992].

[7] D. Kimelman and G. Sang'udi. *Program Visualization by Integration of Advanced Compiler Technology with Configurable Views*. Technical report, IBM T. J. Watson Research Center, September, 1992.

[8] P. Kondapaneni, C. Pancake, and C. Ward. *A Visual Programming Tool for Fortran D*. December, 1992.

[9] D. Reed, *et al. The Pablo Performance Analysis Environment*. Technical report, University of Illinois at Urbana–Champaign, Department of Computer Science, October, 1992.

[10] S. Sarukkai and D. Gannon. *Parallel Program Visualization using SIEVE.1*. Proceedings of the 1992 ACM International Conference on Supercomputing, July, 1992.

[11] S. Srinivas and D. Gannon. *Interactive Visualization and Animation of Parallel Programs*. Technical report (abstract), Indiana University.