Visualization of Distributed Data Structures for High Performance Fortran-Like Languages

RAINER KOPPLER, SIEGFRIED GRABNER, AND JENS VOLKERT

GUP Linz, Johannes Kepler University Linz, Altenbergerstrafse 69, 4040 Linz, Austria;
e-mail: {koppler,grabner,volkert}@gup.uni-linz.ac.at

ABSTRACT

This article motivates the usage of graphics and visualization for efficient utilization of High Performance Fortran's (HPF's) data distribution facilities. It proposes a graphical toolkit consisting of exploratory and estimation tools which allow the programmer to navigate through complex distributions and to obtain graphical ratings with respect to load distribution and communication. The toolkit has been implemented in a mapping design and visualization tool which is coupled with a compilation system for the HPF predecessor Vienna Fortran. Since this language covers a superset of HPF's facilities, the tool may also be used for visualization of HPF data structures.

1 INTRODUCTION

Distributed memory multicomputers are increasingly being used in scientific computing for high-performance calculations as they offer many advantages over shared memory architectures concerning scalability and costs. On the other hand, the programmer is faced with the nontrivial problem of data distribution. Since these architectures lack a global data space, computational data must be distributed among the local memories of the single processors in such a way that good load balancing is achieved and interprocessor communication is kept minimal. Depending on the size and structure of the computational data, resulting data distributions can become very complex and thus difficult to rate.

Received May 1995
Revised February 1996

Scientific Programming, Vol. 6, pp. 115–126 (1997)
CCC 1058-9244/97/010115-12

Since the quality of a data distribution may have a crucial impact on the efficiency of the computation and research in automatic parallelization [1, 2] has not shown satisfactory results up to now, vendors and researchers have introduced language extensions to Fortran 77 and Fortran 90 which allow explicit specification of distribution layouts for data arrays. The advantages of this approach include minor parallelization efforts and increased portability. High Performance Fortran [3] is the most popular representative of these languages as it offers a portable standard and allows graceful migration of codes, i.e., HPF codes can be run on sequential architectures without modifications.

Currently several HPF compilers are available that mainly perform source-to-source translation to Fortran 77 with calls to portable message-passing libraries, but they seldom offer other facilities and tools. From our point of view, the language extensions introduced by HPF are small but powerful and thus pose the need for additional language-specific, sophisticated tools that assist the programmer in applying HPF's facilities successfully.

Following [4], besides performance analysis tools

a well-designed HPF programming environment also includes an interactive data-mapping assistant whose purpose is to choose a suitable data distribution by utilizing knowledge from both the compiler and the user. The use of graphics and visualization is expected to increase the tool's value.

In this article we introduce a graphical tool which provides visualization of HPF array objects such as distributed data arrays and logical processor arrays. In its current form our graphical data distribution tool (GDDT) aims at the Fortran 77 extension Vienna Fortran [5], which can be regarded as one of HPF's predecessors. The tool extracts distributed array objects from Vienna Fortran source texts, evaluates distribution specifications, and creates a range of graphical displays including viewers for data arrays, processor arrays, and diagrams relating to load distribution and communication.

In Section 2 we motivate the advantages of visual support for data distribution and point out the benefits of this approach for important programming issues. Section 3 describes the concepts and facilities of GDDT. The tool implements our concepts by means of a graphical toolkit consisting of exploratory and estimation tools. Furthermore, a clear separation between compile-time and post-mortem support is made. This separation is essential in order to define the scope of visualization data, which affects the tool's capabilities. Section 4 surveys related work and segregates our tool from other research systems by means of a short classification. A concluding general view of design tools for graphical data distribution systems together with a brief view on our future activities concludes this article.

2 VISUAL SUPPORT FOR DATA DISTRIBUTION

In HPF, the programmer specifies data parallelism implicitly by means of data distribution directives. While directives are characterized by a quite simple syntax, from the semantical point of view they have a considerable impact on the parallelization process and involved optimizations performed by the compiler. Distribution directives affect two key issues to be considered with distributed memory computers:

1. Load distribution. A distribution specification assigns each element of a given data array to one or more logical processors. Assuming the owner computes rule [6], all computations on a given array element are performed by its owner. Computational load in a parallel loop can be considered balanced if array elements have been distributed to the processors evenly.
2. Communication between processors. An array distribution implicitly defines communication requirements according to the array's usage in the code. Communication can be kept minimal if the distribution is reconciled with reference patterns given in the code.

It is a well-known fact that optimization of both these criteria involves the solution of NP-complete problems [7]. Thus, the existing variety of research systems focusing on automatic data distribution can only produce suboptimal distributions and it has become obvious that efficient data distribution requires user-supplied knowledge [4]. In connection with HPF, the programmer is confronted with the following tasks in order to contribute to the generation of efficient parallel programs:

3. Choice of a suitable data distribution. According to the previous criteria, this task can be defined more precisely: All data involved in a single computation should be located on the same processor or neighboring processors such that hops and network contention can be avoided whenever possible. Unsuitable distributions cause much communication where the locations of performance bottlenecks cannot be determined from the HPF source text.
4. Location of data parallelism in the code. The chosen data distribution should be exploited by a sensible usage of array operations, intrinsic functions, and parallel loops. In the case of codes with redistributions and realignments, several code sections must be fine-tuned according to the current distribution.

In order to succeed in the above issues the programmer requires knowledge about single-program multiple-data (SPMD) parallelization, compiler techniques, and finally much imagination is also needed. Since it is obvious that HPF source text does not provide enough information to allow sensible estimations, we propose the usage of graphics, visualization, and navigation to make parallelization based on HPF more effective and straightforward. From our point of view, visual support should cover distribution layouts and communication points and volume.

2.1 Distribution Layouts

The layout of array distributions can become very complex and hard to imagine, especially when

the programmer uses alignments or replications in connection with high-dimensional arrays. Showing an array distribution as the correspondence between array elements and logical processors gives valuable information on load distribution. Our approach provides navigation through high-dimensional arrays and information retrieval based on selections of array elements, distribution blocks, and groups of both.

2.2 Communication Points and Volume

Since the location of communication points and the involved communication primitives is determined by the compiler and thus not available from the source text, information concerning these issues should be made accessible to the user by means of source text annotations [8] or separate diagrams. Here, a clear separation between static and dynamic communication aspects must be done.

Communication caused by redistributions and computations on input data is mainly data dependent, thus its amount and effects do not show before runtime. Here we may only obtain a post-mortem visualization based on trace data. By way of contrast overhead through special communication events, such as overlap communication [6] (see Section 3.2, subsection "Estimation Tools"), can be predicted at compile-time. This allows appropriate visualization before the program run.

We consider visual support at compile-time most effective due to the increase of productivity, i.e., the program tuning cycle is shortened significantly by eliminating unnecessary execution steps. Our concepts cover both compile-time and post-mortem visualization and concentrate on overlap communication and redistributions respectively.

3 OVERVIEW OF GDDT

In order to rate the value of graphics and visualization for the task of data distribution, we have developed a software tool which covers most of the issues discussed in the previous section. It implements individual concepts and techniques and shows our preliminary results regarding graphical-aided data distribution. Figure 1 gives a global overview of the tool's look and feel. Concerning the input language used, the facilities of our GDDT are currently based on Vienna Fortran [5], one of HPF's predecessors. We chose this data-parallel Fortran extension due to the following reasons:

1. Major contributions to HPF come from Vienna Fortran. Roughly speaking, Vienna Fortran's language extensions for data-parallel programming can be regarded as a superset of those provided by HPF. Thus, all facilities provided by GDDT can also be applied in context with HPF. Furthermore, language-specific parts of the tool's implementation are encapsulated into well-defined components in order to simplify future adaptions, primarily to HPF.
2. The tool is tightly coupled with the Vienna Fortran Compilation System (VFCS) [9], in particular with the Vienna Fortran compiler, for the purpose of information exchange. The VFCS features an ideal information source for GDDT by exporting basic information for visualization such as distribution specifications, overlap descriptions (see Section 3.2, Subsection "Exploratory Tools"), and communication points.

The following two sections give an overview of the graphical facilities provided by GDDT. At first we survey the extent of information available for visualization where a clear distinction between compile-time and post-mortem information is necessary. Based on this classification, two interacting suites of graphical displays and tools are introduced in the sequel.

3.1 Visualization Data

We classify basic data for visualization into static and dynamic data. In our context, the term *static* characterizes data which can be collected before the program run, while *dynamic* data are available only during or after run-time.

The most important kind of static data is a Vienna Fortran compilation unit consisting of several subroutines, functions, and a main program. Definitions of processor shapes and distributed data arrays and DISTRIBUTE statements are of major interest for visualization. There are two ways for GDDT to obtain the contents of a compilation unit:

1. Vienna Fortran source text. If only the source text is available, distributed data arrays and related information can be extracted from it by means of a rudimentary parser. The source text may be shown in a built-in text editor during visualization where feedback between displays and corresponding source text locations is permanently maintained (see Fig. 1).
2. Vienna Fortran compiler. Each compilation unit that has been loaded into the VFCS and been

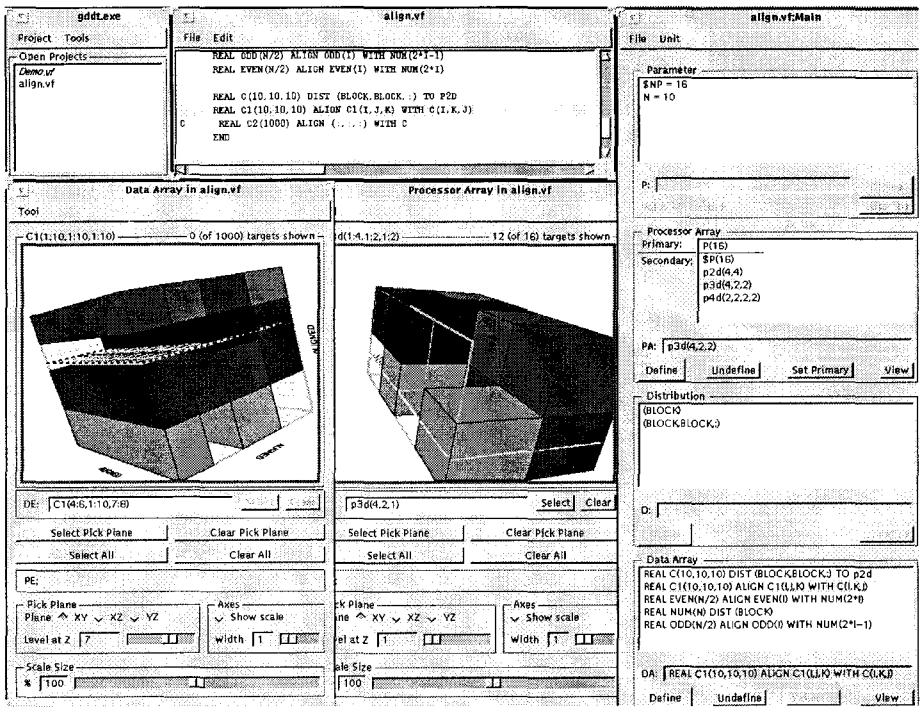


FIGURE 1 Overview of GDDT.

completely parallelized by the compiler can be exported to GDDT by means of its internal representation. Here no additional parsing is necessary.

In each case the obtained data is translated to an internal data structure model which forms the basis for all further operations. The model is displayed in the data structure editor (DSE) as shown in Figure 2. The editor collects parameters, processor shapes, distribution types, and data arrays in separate panels and allows arbitrary, interactive changes to their definitions. Furthermore, additional symbol definitions can be specified. The input's correctness is enforced by checks based on the language definition.

The contents of a given compilation unit together with all editing operations performed on it by means of the DSE form a project that can be saved, loaded, and printed. GDDT can manage multiple projects at the same time.

Besides compilation units, static visualization data include information produced by the compiler, in particular transformation and optimization information. Communication aspects such as location and type of communication are of major interest for visualization. Currently only overlap descriptions are utilized by our tool. Support for detailed static or dynamic communication information, that will be gained from node programs or trace data, respectively, is the subject of our future work.

Visualization of post-mortem information has been extensively investigated in a number of research systems [10]. Most existing work is based on trace data, which consist of various information recorded during the program run. Trace data are evaluated with respect to different aspects such as algorithms [11], computational data [12], interconnection topology [13], or performance [14]. In our work, we concentrate on visualization of redistributions and realignments of data arrays together with the impact of such events on the overall execution time. Hereby dynamic visualization data cover timings of redistributions and realignments causing data migration together with their locations in the source text. GDDT utilizes such data, which are provided by the VFCS, in order to replay migration of data arrays.

3.2 Visualization Toolkit

GDDT provides a set of visualization tools which allow exploration of graphical array representations and diagrams by means of a few point-and-click operations. The major aims of the toolkit are to assist the user during exploration of large arrays with complex distri-

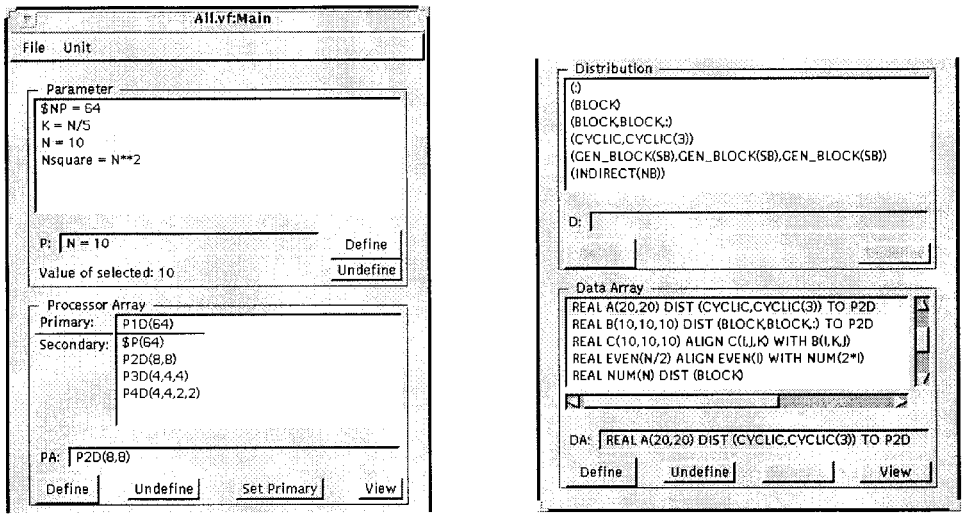


FIGURE 2 Data structure editor.

butions and to provide appropriate graphical interpretations derived from data concerning load distribution and communication. Following these aims we classify GDDT's facilities into two interacting groups: exploratory tools and estimation tools.

Exploratory Tools

This group covers both static and dynamic data and concentrates on facile navigation within distributed data arrays and logical processor arrays emphasizing mapping relationships between data and processors.

Exploration of Mappings. Visualization of array structures plays an important role here. GDDT displays arrays with arbitrary numbers of dimensions where up to three dimensions can be shown simultaneously and remaining dimensions are hidden. Figure 3 shows examples of array representations for each rank. The hatched area within array *A3D* is called the selection plane. Since selection of array elements in three dimensions may be ambiguous due to loss of information during projection onto the two-dimensional display screen, a two-dimensional slice is provided in order to enable unambiguous selections. The plane may be positioned in parallel to the XY, XZ, or YZ plane and moved along the third axis by means of a slider (Fig. 4).

In most cases the user cannot perceive characteristics of distribution layouts directly in the global representation so navigation tools are required to perform basic view manipulation. For this purpose GDDT offers primitives for translation, zooming, and rotation. Control of the tools conforms to the AVS environment [15], so users familiar with this system can easily utilize GDDT's facilities.

Data arrays and processor arrays are shown in separate windows called array viewers. Array viewers can be opened by selecting arrays in the DSE window and pressing the *View* button. Currently only one data array viewer and one processor array viewer can be shown simultaneously. It must be noted that data and processor arrays can be combined separately, i.e., it is possible to show a data array's distribution on another processor array shape than the one given in the distribution specification.

When the user has opened data array and processor array viewers and adjusted the representation according to his or her needs, the distribution can be explored by means of selections. Figure 4 shows relationships between a three-dimensional data array and a two-dimensional processor array.

The data array viewer shows the structure of a particular distributed data array. The user selects array elements in order to know how they are mapped to logical processors. Nearest-neighbor mappings can be verified by evaluating the mappings of adjacent elements. Whenever an element is selected, a colored cube (or a square in two dimensions) appears and the processor array viewer highlights all logical processors which own the element in the same color. We refer to the data element as the *source*, the owning processors are called *targets*. In Figure 4 on the left two arbitrary, neighboring elements—named C(17, 12, 1) and C(18, 12, 1)—have been selected. Since they are located on the same processor (*P2D*(7, 4)), the two sources and their target are highlighted in the same color.

The processor array viewer shows the shape of a

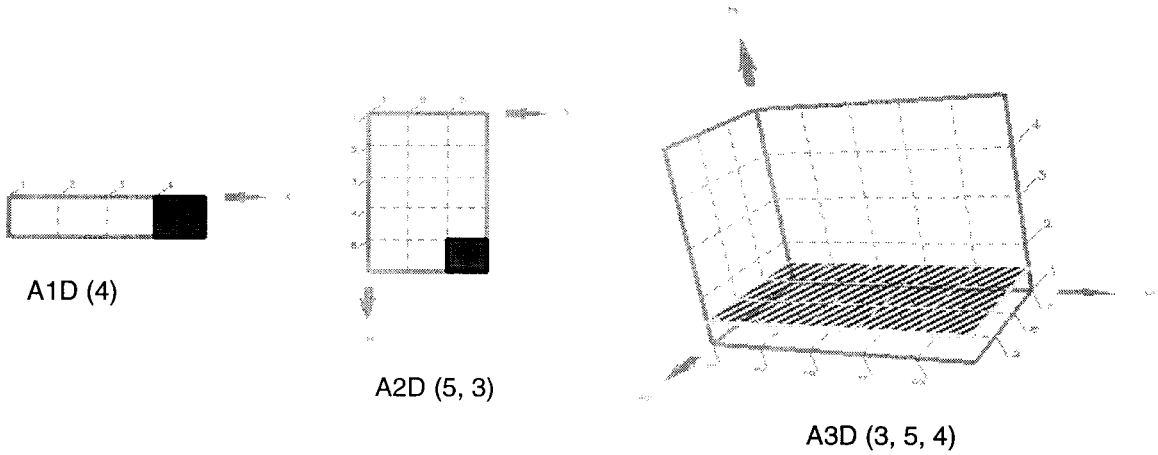
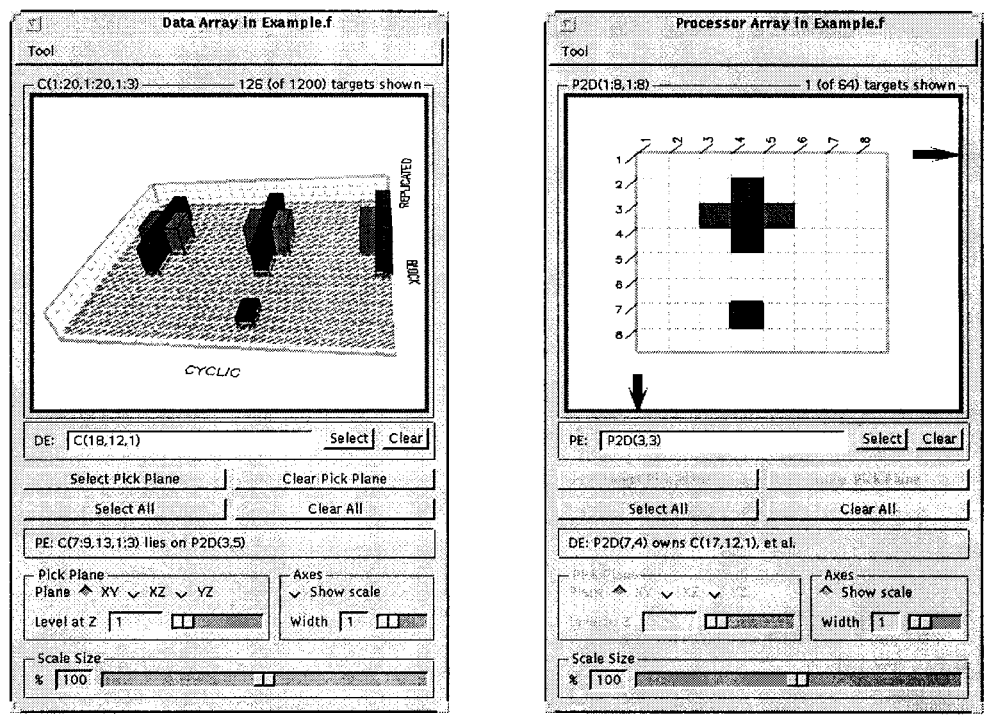


FIGURE 3 Graphical representation of arrays.

particular logical processor array and provides the same facilities as the data array viewer. The selection of a logical processor lets the data array viewer highlight all data blocks assigned to this processor. This facility shows characteristic patterns for different distributions. In Figure 4 on the right processor, *P2D*(3, 4) and its four direct neighbors have been selected. The shape of these selections survives in the target display, indicating that only regular distribution functions have been used. Mappings with preserved shapes are especially suitable for algorithms where calculations on array elements require values from neighboring elements, e.g., relaxation methods. Characteristic patterns are given in Figure 4 as follows:



REAL C(20,20,3) DIST(BLOCK, CYCLIC, :) PROCESSORS P2D(8,8)

FIGURE 4 Exploration of a distribution.

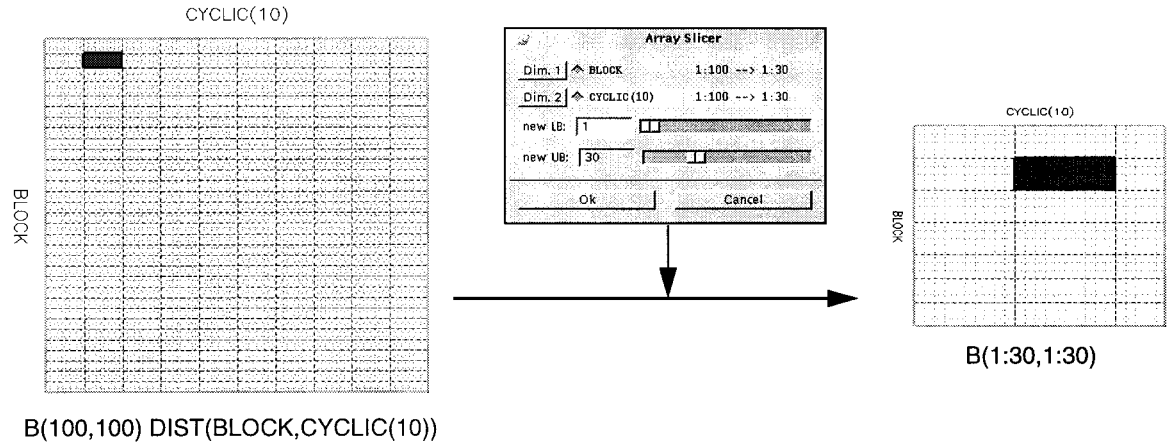


FIGURE 5 Array slicing facility for large arrays.

1. The cyclic distribution along the Y axis is characterized by three replications of the target shape (horizontally).
2. The block distribution appears as an enlargement of the target shape in the X direction (vertically).
3. The replication of the two-dimensional distribution along the Z axis is indicated by means of bars (ascending from the selection plane).

While elementwise selection might suffice for small arrays, it is not reasonable for large data arrays or processor arrays that represent massively parallel systems. In order to promote scalability of the exploration facilities, multiple selection levels are provided:

1. Single-element selections via mouse clicks or textual specification.
2. Multielement selections via "rubber-banding" or textual specification by means of slices.
3. Distribution block selections via mouse clicks (see Fig. 5).
4. Plane selections in 3D (*Select Pick Plane* in Fig. 4).
5. Global selections (*Select All* in Fig. 4).

Furthermore a slicing facility for the graphical representation is needed in order to allow detailed investigation of arbitrary array portions. Figure 5 shows an example of array slicing. For the purpose of simplicity, the control elements of the data array viewer before and after slicing were omitted.

Display of Overlap Areas. Overlap areas [6] are a means to optimize communication in connection with regular computations on data arrays whose dimensions are distributed blockwise. Each block can be associated with an overlap area which denotes the smallest rectilinear contiguous area around the block containing all nonlocal elements being accessed by the block's process. The extent of the overlap area is determined from an overlap description. Each overlap description is specific to one single array and computed by the compiler during parallelization by means of the array's distribution and reference patterns given in the code.

The overlap description is made available to GDDT together with the array's distribution. When the block-selection level is chosen, as shown in Figure 6, a block's overlap area can be shown by means of a key-click combination. Furthermore, the overlap description can be modified and returned to the VFCS.

Replay of Dynamic Distributions. Arrays whose distribution may change at run-time are called dynamic in the terminology of HPF and Vienna Fortran. Promising uses for such distributions include dynamic load balancing for codes that are executed on different architectures or where the size or structure of computational data varies. A good example for the latter case are particle-in-cell codes [16].

In order to provide an insight into the amount of redistributions performed during the program run, the Vienna Fortran Engine [17] records all events relating to data migration and forwards them to GDDT. The distribution sequence can then be replayed by the animator tool. If the source text is available to GDDT, associations between events and corresponding lines of the source text are shown.

Figure 7 depicts a very simple distribution sequence. Currently only source-based distribution sequences are shown by the tool, i.e., traces which have

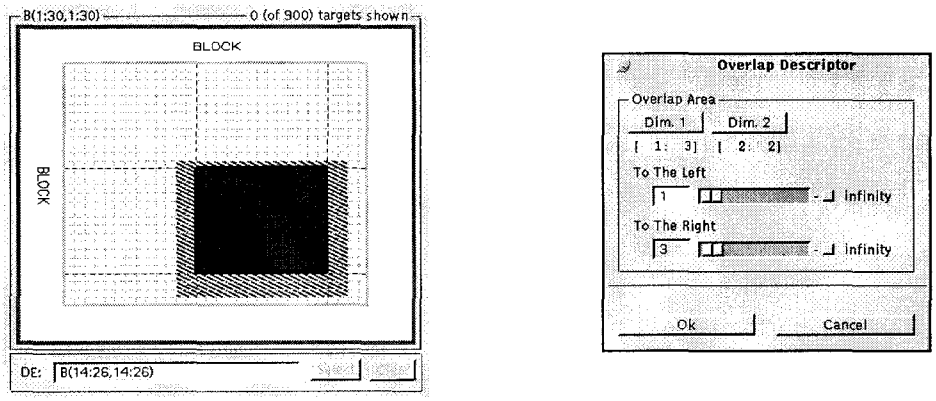


FIGURE 6 Display and modification of overlap areas.

been gained from the source text by consecutive execution of redistribution statements. Sequences derived from actual trace data require sophisticated measurement. Facilities for their generation are expected to be available in the future.

Regarding Figure 7, it must be noted that the executable directives REDISTRIBUTE and REALIGN of HPF are both subsumed by the DISTRIBUTE statement in Vienna Fortran. The animator tool shows time, location, and specification of dynamic distributions, provides controls for stepwise and continuous replay, and allows the user to exclude events from replay, which may be useful for source-based sequences gained from source texts containing conditional statements.

Estimation Tools

From our point of view, the main purpose of visualization in the area of scientific parallel computing is to reduce large amounts of complex data to meaningful displays and so to facilitate perception of patterns, relationships, and anomalies. In context with HPF, such displays should assist programmers to make clear decisions between alternative data distributions, assuming that they know how to interpret the graphical representation. As soon as a tool lets the programmer recognize a possibility to improve a distribution, it has proven to be useful.

With GDDT, estimations are guided by diagrams relating to the key issues of load distribution and communication. Since we concentrate on compile-time estimations at the moment, the information base consists of static data including HPF data structures and overlap descriptions. In order to round out the usefulness of the visual toolkit, the user may return changes to the information base to the VFCS, where they can be utilized for further analysis or for code generation.

Load Distribution. For each data array, GDDT can create a two-dimensional chart called a data load diagram which illustrates how evenly the array has been distributed among all logical processors. Figure 8

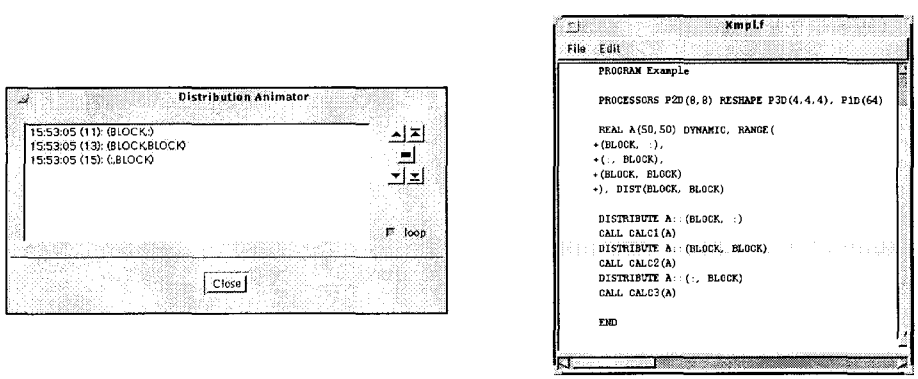


FIGURE 7 Replay of dynamic distributions.

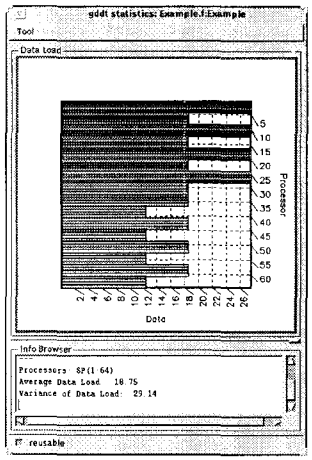


FIGURE 8 Display of load distribution.

shows a data load diagram for array C from Figure 4. For each processor, a bar depicts how many array elements have been mapped to it with respect to the maximal processor load. The coloring tries to simplify the distinction of single bars, especially for the case of large processor numbers. Here the representation can also be sliced with respect to processors.

The info browser below shows statistical information, e.g., absolute element numbers if the user selects a single bar. Load imbalances as given in Figure 8 characterize regular mappings to two-dimensional processor arrays where processors with small index values receive more elements than others.

Our experience has shown that the restricted set of distribution functions provided by HPF hardly produces serious load imbalances. By way of contrast concerning Vienna Fortran, the availability of such tools has been proven valuable since irregular distribution functions (blocks of arbitrary sizes or distribution by means of mapping arrays) may produce distributions which are hard to understand.

Overlap Communication. Overlap areas around distribution blocks increase the efficiency of local computations based on regular patterns. Communication resulting from updates of overlap regions is determined by the compiler and depends on the chosen overlap description. During the update phase, each processor receives contributions to the overlap area of its block from the owning processors of adjacent blocks. Since the programmer can neither control the computation of overlap descriptions nor estimate the impact of overlap communication on the overall execution time, display of overlap communication shows inefficiencies resulting from unsuitable data distributions or ill-chosen overlap descriptions in an early phase of development.

Overlap communication between pairs of processors is shown in GDDT by means of a three-dimensional chart called overlap communication display. For each processor pair (Sender, Receiver) a bar indicates the volume of data elements to be exchanged. Also here the coloring helps the user to distinguish single bars. In order to cope with the large amount of information, the display can be zoomed, rotated, translated, and sliced. Bars can be selected to obtain detailed information about communication. Displays based on regular distributions (as shown in Fig. 9) exhibit characteristic stencils, which show differences with respect to the involved overlap area widths, distribution functions, replications, and the processor array rank. Stencils with equal bar heights (as shown in Fig. 9) depict well-balanced overlap communication. By way of contrast, ill-chosen overlap descriptions or block distributions with varying block sizes (possible in Vienna Fortran) may produce bars with different heights, warning the user of imbalanced communication.

4 RELATED WORK

Compared to other activities occurring during parallel software development—such as process mapping, performance analysis, and debugging—data distribution has not been supported adequately by visual tools yet. Furthermore, the lack of a taxonomy for existing research tools often causes confusion with respect to their applicability. In order to rate the contribution of our work, we classify some well-known systems for data distribution visualization according to the data structure level they operate on.

1. Problem-related structure. This class of tools visualizes mappings of scientific data structures such as meshes and matrices. Data distribution is performed automatically by partitioning algorithms and mainly aims at selected target architectures. Most popular representatives of this class are DecTool [18], GraphTool [19], and DDT [20]. MATRIX [21] also visualizes runtime states.
2. Computational structure. Tools that operate on this level are associated with a concrete programming language. Existing work mainly concentrates on run-time behavior of data structures. Well-known representatives are VIPS [22] for dynamic data structures in Ada and VISTA [12], which shows contents of program

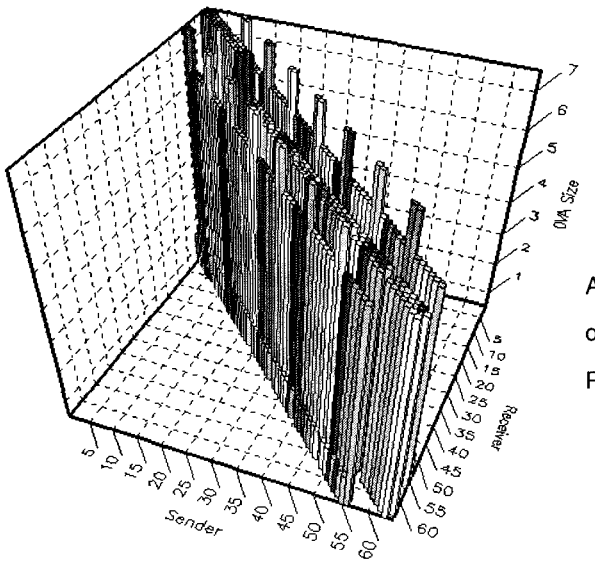


FIGURE 9 Display of overlap communication.

variables during execution. Recent work also focuses on HPF and addresses both source-based [23] and trace-based [24] visualization.

Concerning this classification, GDDT belongs to the latter class. Unlike most existing systems, GDDT focuses on compile-time visualization because of productivity reasons. Furthermore, all systems are associated with one programming language, while GDDT may be used for the whole family of HPF-like languages due to its modularity.

Kimelman et al. [24] report on a set of high-level views for visualization of the execution of HPF programs. Mapping relationships between data arrays and processors are displayed similar to GDDT but just up to two dimensions. DDV [23] shows processor utilization in term of data references. Visualization is based on source text and additional assumptions and thus produces vague results.

GDDT's facility to display overlap communication approaches that of ParaGraph [13]. ParaGraph's communication matrix display features a generalization of GDDT's overlap communication display as it shows all communication between pairs of processors. It also provides an animated display based on trace data, whereas GDDT's display is entirely based on compile-time information.

5 CONCLUSIONS

In this article we motivated visual support for efficient utilization of HPF's data distribution facilities. Our approach proposes a graphical toolkit consisting of exploratory and estimation tools that assists the programmer in designing and rating data distributions. We presented a software tool which implements our concepts and showed several opportunities where visualization and navigation can contribute to the understanding of complex distributions.

An important design issue is the separation of static and dynamic concerns. On the one hand, it specifies to which extent the user can contribute to generation of efficient parallel programs by means of graphical operations. On the other hand, it defines the amount of information available to the user for graphical evaluation. In each case, interaction with external systems such as compilers or execution systems is necessary in order to create meaningful displays and to obtain sensible ratings.

Summarizing our experiences with GDDT and following [10], we define the design goals for graphical systems with focus on data distribution as follows:

1. Ease of understanding. The user's ability to perceive distribution patterns, mapping relationships, and anomalies in distribution, data load, or communication should be promoted by means of appropriate displays.
   A. Besides displays for distributed data structures a separate display for the target architecture should be provided.
   B. Primitives for basic view manipulation (including rotation, zooming, and slicing) are essential in order to choose suitable settings for large or complex data structures.

   C. Relationships between data of various detail and processors shall be stressed by means of correlative linking, e.g., by using equal colors.
   D. The user should be able to temporarily intersperse graphical representations with additional information concerning transformation and optimization, e.g., overlap areas around data blocks or communication points within the source text.
2. Ease of usage. This issue is particularly important for exploratory tools as here the user does not know a priori which setting will promote understanding.
   A. The user should be able to carry out basic view manipulations quickly by means of mouse operations or key-mouse combinations.
   B. Several selection levels should allow the user to choose at which level of detail a distributed data structure shall be explored (single elements, slices, distribution blocks, groups of elements or blocks).
   C. Data distribution systems should be seamlessly integrated into compilation systems and so produce environments that provide permanent associations between source code, data structure mapping, and estimation displays.
3. Portability. Data distribution systems should not depend on a particular target architecture. Systems aiming at HPF-like languages fulfill this requirement implicitly since data are distributed to virtual topologies. Concerning such systems, flexibility is an essential property in order to provide support for some important predecessors and derivatives of HPF and to cope with future language extensions.

For the future, we plan adaptation of GDDT to HPF and increased utilization of communication information provided by the compiler. In order to visualize HPF arrays, an additional parser will be integrated which handles HPF's syntactical characteristics and creates an internal model based on Vienna Fortran's equivalent facilities. Regarding visualization of communication we plan to incorporate timings and sophisticated source-text associations.

ACKNOWLEDGMENTS

This work is partially supported by the Austrian Ministry of Science, Research, and Art (BMWFK) project CEI-PACT, subproject Work Package 1: "Advanced Compiler Technology."

AVAILABILITY

GDDT is available on the following platforms: SunOS, IRIX, HP-UX, and Linux. The SunOS binary can be obtained from `prometheus.gup.uni-linz.ac.at:/pub/gddt` via anonymous FTP. All illustrations in this article can also be obtained from this site as SunRaster images.

REFERENCES

1. M. Gupta and P. Banerjee, "Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers," *Computer*, vol. 3, pp. 179–193, 1992.
2. K. Knobe, J. Lukas, and M. Weiss, "Optimization techniques for SIMD Fortran compilers," *Concurrency Practice Exp.*, vol. 5, pp. 527–552, 1993.
3. High-Performance Fortran Forum. "High performance Fortran language specification version 1.0." *Sci. Prog.*, vol. 2, pp. 1–170 Spring/Summer 1993.
4. V. Adve, A. Carle, E. Granston, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, J. Mellor-Crummey, S. Warren, and C. W. Tseng, "Requirements for data-parallel programming environments," *Parallel Distrib. Technol.*, Vol. 2, No. 3, pp. 48–58, 1994.
5. H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald, "Vienna Fortran—A language specification, Version 1.1," Department of Statistics and Computer Science, University of Vienna, Tech. Rep. ACPC/TR 92-4, 1992.
6. H. Zima and B. Chapman, "Compiling for distributed-memory systems," Department of Statistics and Computer Science, University of Vienna, Tech. Rep. ACPC/TR 92-17, 1992.
7. M. Garey, D. Johnson, and L. Stockmeyer, "Some simplified NP-complete graph problems," *Theor. Comput. Sci.*, Chapter 1, pp. 237–267, 1976.
8. D. McCallum and M. Quinn, "A graphical user interface for data-parallel programming," in *Proc. of HICSS-26*, pp. 5–13, 1993.
9. B. Chapman, S. Benker, R. Blasko, P. Brezany, M. Egg, T. Fahringer, H. Gerndt, J. Hulman, B. Knaus, P. Kutschera, H. Moritsch, A. Schwald, V. Sipkova, and A. Zima, "Vienna Fortran compilation system—Version 1.0—user's guide," Department of Statistics and Computer Science, University of Vienna, 1993.
10. G. Tomas and C. Ueberhuber, *Visualization of Scientific Parallel Programs*. Berlin, Heidelberg: Springer-Verlag, 1994.
11. M. Brown and R. Sedgewick, "*A system for algorithm animation*," *Comput. Graphics*, Vol. 18, pp. 177–185, 1984.

12. A. Tuchman, D. Jablonowski, and G. Cybenko, "A system for remote data visualization," Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, CSRD Report 1067, 1991.
13. M. Heath and J. Finger, "ParaGraph: A tool for visualizing performance of parallel programs," University of Illinois and Oak Ridge National Laboratory, Tech. Rep. ORNL/TM-M813, 1994.
14. A. Malony, D. Hammerslag, and D. Jablonowski, "TRACEVIEW: A trace visualization tool," *Software*, Vol. 8, pp. 19–28, 1991.
15. Advanced Visual Systems, *AVS—User's Guide, Release 4*, Waltham, MA: AVS, 1992.
16. G. Fox, "Hardware and software architectures for irregular problem architectures," in *Unstructured Scientific Computation on Scalable Multiprocessors*, Boston, MA: MIT Press, pp. 125–160, 1992.
17. B. Chapman, P. Mehrotra, H. Moritsch, and H. Zima, "Dynamic data distributions in Vienna Fortran," in *Proc. Supercomputing '93*, pp. 284–293, 1993.
18. N. Chrisochoides, E. Houstis, and J. Rice, "Mapping algorithms and software environments for data parallel PDE iterative solvers," *J. Parallel Distrib. Comput.*, vol. 21, pp. 75–95, 1994.
19. S. Hammond, "Mapping unstructured grid computations to massively parallel computers," PhD thesis, Department of Computer Science, Renesselaer Polytechnic Institute, New York, 1991.
20. N. Floros and J. Reeve, "Domain decomposition tool—An abridged user's guide," Department of Electronics and Computer Science, University of Southampton, Tech. Rep. CAMAS-TR-2.2.2.8, 1994.
21. R. Paul and D. Poplawski, "Visualizing the performance of parallel matrix algorithms," in *Proc. 5th Distributed Memory Conf.*, Vol. 2, pp. 1207–1212, 1990.
22. S. Isoda, T. Shimomura, and Y. Ono, "VIPS: A visual debugger," *Software*, Vol. 6, pp. 8–19, 1987.
23. S. Hackstadt, and A. Malony, "Data distribution visualization for performance evaluation," Department of Computer Science, University of Oregon, Tech. Rep. CIS-TR-93.21, 1993.
24. D. Kimelman, P. Mittal, E. Schonberg, P. Sweeny, K. Wang, and D. Zernik, "Visualizing the execution of high performance Fortran programs," IBM Thomas J. Watson Research Center, Tech. Rep., 1994.