Advanced Visualization and Data Distribution Steering in an HPF Parallelization Environment[*]

P. Brezany, P. Czerwiński and K. Sowa
Institute for Software Technology and Parallel Systems
University of Vienna
Liechtensteinstrasse 22, A-1090 Vienna, Austria
{brezany, przemek, sowa}@par.univie.ac.at

R. Koppler and J. Volkert
GUP Linz, Johannes Kepler University Linz
Altenbergerstrasse 69, 4040 Linz, Austria
{koppler, volkert}@gup.uni-linz.ac.at

Abstract

On distributed-memory systems, the quality of the data distribution has a crucial impact on the efficiency of the computation. Sophisticated visualization of large in-core and out-of-core data, and steering capabilities of the debugging system significantly reduce program development cycle, especially for *irregular applications*. In this paper we present an advanced system for visualization and steering of data distributions based on the Graphical Data Distribution Tool (GDDT) and the advanced HPF symbolic debugger SPiDER. The development focused on efficient support for regular and irregular data distributions and brings several significant contributions: (1) on-line visualization of distributions and values of large in-core and out-of-core data structures, (2) quality control of data distributions using (re)distribution animation, and (3) dynamic data redistribution in parallel programs.

1 Introduction

Large-scale parallel applications, which typically use huge amounts of data, require special support for manipulation of global entities of the program. Since program performance on distributed-memory systems significantly depends on data distribution, the developer needs support for sophisticated visualization of large in-core and out-of-core data[1], and for steering data distributions. In particular, developers of *irregular applications* greatly benefit from on-line visualization and steering capabilities of the programming environment, since in such applications the distribution of major data arrays and their access patterns are known only at run time.

Despite of many activities in parallel software development – such as process mapping, performance analysis, debugging – data distribution has not been supported

[*]This research is being carried out as part of the research project "Visualization in Parallelization Environments" (grant P11157-TEC) and Special Research Program SFB F011 "AURORA" supported by the Austrian Science Foundation (FWF)

[1]Out-of-Core data structures are too big to be kept in main memory, therefore they are stored on disks.

adequately. Most existing systems for visualization of data distribution (e.g., *HPF-Builder* [8]) are restricted to compile-time evaluation of data because of productivity reasons. Visualization and steering of dynamic data distribution still suffer from the lack of completeness of support. One of the most robust systems in this field is *DAQV* [6], which has been designed for HPF compilers. It provides a framework for accessing and modifying data at the run time in order to simplify visualization and computational steering. The importance of distributed data visualization has been recognized in data-parallel debuggers. *Prism* [1] provides various displays based on histograms, multidimensional graphs, surfaces and vector representations of program's data including support for data-parallel programs written in CM-Fortran. Most existing HPF debuggers (e.g., *PDT* [5]), provide a global data visualization for viewing entire arrays and array segments allocated across processors.

So far, visualization has been concentrated more on a convenient, easily-interpreted representation of data. However, combining sophisticated facilities of the visualization tool with program execution control, data inspection, and distribution steering capabilities of the debugger gives the user a robust system for debugging and tuning programs. This paper presents an advanced system for visualization and steering of data distributions based on the Graphical Data Distribution Tool (GDDT) [7] and SPiDER – an advanced HPF symbolic debugger [3]. The development focused on efficient support for regular and irregular data distributions and brings several significant contributions:

- on-line visualization of distributions and values of large in-core and out-of-core data structures by means of efficient parallel I/O software,
- quality control of data distributions at run time and program tuning using an animation facility, which replays the history of data distributions and redistributions with respect to data migration, and
- dynamic data redistribution in a program after stop at a breakpoint during a debugging session.

The rest of this paper is organized as follows. Section 2 outlines our approach and presents the software architecture and the capabilities of system. Support for debugging in the compiler and the run-time system, and the implementation of interfaces are discussed in Section 3. We conclude in Section 4.

2 System Design

2.1 Components and Interfaces

The overall system consists of the HPF debugger SPiDER and the visualization tool GDDT. Both tools are loosely coupled. Up to now GDDT has offered communication with compilers and run-time systems via a socket-based data exchange protocol. In order to control GDDT from a debugger, the tool was extended by an additional text-based command interface. Whenever the programmer requests visualization of a distributed array, its distribution, or its values, the debugger sends a command to GDDT. The interprocess communication interface used allows remote steering, for example, opening and initialization of *Data Structure Editor* (DSE) windows using data gained from the interface, or opening of data array viewers. In order to let GDDT visualize a snapshot of the program, the debugger first converts the declarative part of the program into source form and issues an *open* command. This command causes GDDT to open a new DSE window. The program instrumented for debugging creates an additional file holding distribution histories of dynamically distributed arrays. On the user's request the debugger sends the name of this file to GDDT using the *add* *distributions* command. While the *open* command is issued once, the *add distributions* command is issued during each visualization request. Whenever the programmer requests visualization of the contents of an array, the debugger calls a run-time function that stores the array contents in a parallel file and passes the name of the array file on to GDDT using the *add values* command. GDDT accesses the file using parallel I/O and extracts arbitrary slices from it. This approach covers both in-core and out-of-core data.

2.2 Data Distribution Steering

In long-running applications the user should be able not only to inspect the program state at a given breakpoint but also to verify the data distribution quality and to change data distributions. Changing the distribution of an array during program execution may cause loss of the program consistency. As a result, the program may crash or produce wrong results. Loss of consistency may occur when the user redistributes an array that the compiler assumed to have an invariant distribution at this place or somewhere later in the program the distribution specified in the source code is expected. The compiler can perform various optimizations based on the knowledge of distributions, producing code which is not transparent to an unexpected redistribution. In this section we specify conditions, under which an on-line redistribution is safe.

In HPF only arrays with the `DYNAMIC` attribute can be redistributed. The `DYNAMIC` attribute tells the compiler to generate code for this array that is distribution transparent, i.e., its behavior does not depend on the distribution of arrays and there are no distribution-driven optimizations. HPF also imposes that any array which is explicitly aligned to another array cannot be redistributed. On the other hand, the redistribution of an array triggers redistribution of all arrays aligned to it.

Other limitations depend on the compilation strategy used in the given HPF compiler. An example of such a situation is an independent parallel loop (i.e., having `INDEPENDENT` attribute) compiled using inspector/executor strategy [2]. In the inspector phase communication schedules and iteration sets are computed according to the distribution of arrays used in the loop, and in the execution phase the schedules are used for data transfer among processors to enable fully parallel execution of the loop. Changing the distribution of arrays inside of the parallel loop makes useless all data precomputed in the inspector phase thus renders any further correct execution of the loop totally impossible.

All these conditions can be checked by the debugger using information obtained from static code analysis performed by the compiler. Unfortunately, compile-time code analysis is not sufficient when the programmer uses the `REUSE` clause or schedule variables. These tell the compiler that results of the computation of communication schedules can be preserved and reused in further executions of the loop to skip the time-consuming inspector phase. In this situation, changing the distribution of any array invalidates all schedules (both contained in schedule variables and implicitly used in `REUSE` clauses). Utilization of an invalid schedule leads to erroneous program behavior or a program crash. Decision about reusing given schedule variables can be made dynamically, so it is not possible, only through static analysis, to determine whether performing redistributions in the given program state is safe or not. So far, the problem remains unsolved and the current version of the system does not address it. Full solution requires additional support in run-time system and extension of the debugging system. We plan to address this problem in the future.

2.3 Data Visualization

GDDT provides facile navigation within distributed data arrays and logical processor arrays emphasizing mapping relationships between data and processors. GDDT displays arrays with arbitrary numbers of dimensions where up to three dimensions can be shown simultaneously. Data arrays and processor arrays are shown in windows called *array viewers*. On the user's request separate viewers for each distribution of a data array can be opened, which allows the user to compare several distributions graphically. It is also possible to show a data array's distribution on another processor array shape than the one given in the distribution specification. The processor array viewer shows the shape of a particular logical processor array. The selection of a logical processor lets the data array viewer highlight all data blocks assigned to this processor. Besides exploratory displays, GDDT provides also displays for evaluation of data distributions, for example, a *data load diagram*.

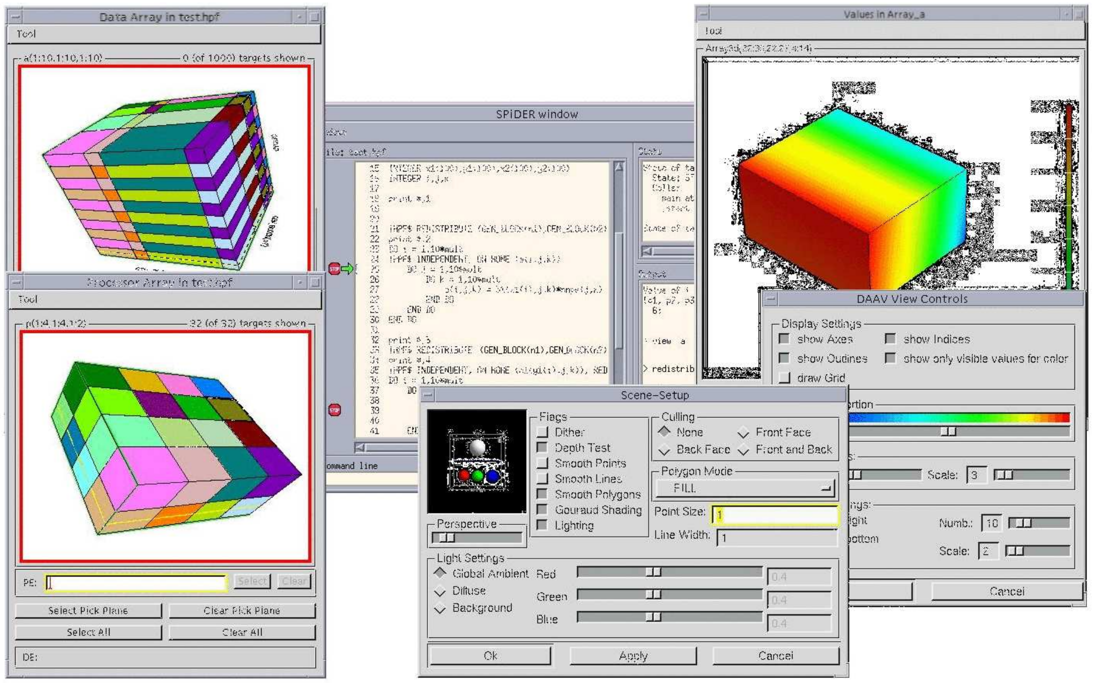
Regarding visualization of post-mortem information GDDT concentrates on redistributions of data arrays. Visualization data cover redistributions and their locations in the source text. The HPF run-time system compiles all distribution changes and forwards them to GDDT. The user can see an animated replay of a redistribution sequence by opening a data array viewer and selecting the *animator tool*. The tool offers stepwise or continuous replay. The user can observe the migration of arbitrary array elements by selecting them in the data array viewer. At each redistribution the corresponding targets in the processor array viewer change their location, showing how array elements migrate from one set of processors to another.

While it is useful to inspect data distributions at run time, programmers usually use debuggers to inspect the contents of data arrays. Thus, GDDT provides an additional facility called *data array value viewer*. This viewer visualizes array values by mapping them to colors within a spectrum between blue and red. Low values are associated with "cool" colors such as blue, whereas high values are highlighted using "hot" colors such as red. Since small deviations between two snapshots can be hardly detected by looking at two similar pictures, GDDT also allows to visualize differences between snapshots. For this purpose, it provides a *contents file manager*, which holds up to seven snapshots and allows the user also to choose pairs of snapshots in order to show deviations between them. According to the above coloring scheme strong deviations are highlighted in red.

The data array value viewer does not restrict visualization to the array shape inherited from the data array viewer. Sometimes the user might want to clip pieces of a three-dimensional array in order to examine its interior. This can be achieved by specifying a sub-shape of the inherited shape. Whenever an appropriate slice has been found, it can be explored using scaling, translation, and rotation. Figure 1 shows an example debugging session where execution stopped at a breakpoint and the user requested visualization of the distribution and the values of the three-dimensional array `a`. The display shows a GEN_BLOCK/GEN_BLOCK/CYCLIC distribution.

3 Implementation

At the program level, the steering capability of the system requires an efficient implementation of a redistribution function in the run-time system. At the user level, we added a new debugger command which allows to change a distribution of the given array after stopping at a breakpoint. The debugger is responsible for checking the correctness of the redistribution at the given point in the program, using information generated by the compiler, stored in a symbol table file [4]. If the redistribution of the given array is safe, the debugger feeds the new data distribution specifications into the run-time system and executes the redistribution function. This function is defined



```
INTEGER x1(100),y1(100),x2(100),y2(100)
INTEGER i,j,k

print *,1


!HPF$ REDISTRIBUTE (GEN_BLOCK(n1),GEN_BLOCK(n2)
print *,2
DO i = 1,10*mult
!HPF$ INDEPENDENT, ON HOME (a(i,j,k))
    DO j = 1,10*mult
        DO k = 1,10*mult
            a(i,j,k) = [illegible](i,j,k)*[illegible](j,k)
        END DO
    END DO
END DO

print *,3
!HPF$ REDISTRIBUTE (GEN_BLOCK(n1),GEN_BLOCK(n2)
print *,4
!HPF$ INDEPENDENT, ON HOME (a(gi(i),j,k)), RED
DO i = 1,10*mult
    DO
```

Figure 1: Example of a debugging session with SPiDER.

in a module visible within a global scope of the program and linked with it.

A similar approach is used for data value visualization. On the user's request to display the contents of an array, the debugger calls a special subroutine in the program that stores the array contents or a section thereof in a parallel file, and passes the file name on to GDDT. The current implementation of parallel files is based on MPI-IO. Apart from the values of elements, the file contains metadata that are necessary for interpretation of its contents, i.e., the array rank, shape, size, and element type. The library executes I/O operations in two phases. First, a pair of complex MPI data types is prepared, one for the file layout and another for the memory layout. They describe a range of elements the operation affects and constitute the correspondence between elements in file and memory. Such a pair can describe the whole, even quite complex I/O operation that must be done on one I/O node. In the second phase, one collective call to `MPI_File_Write` or `MPI_File_Read` is made. This strategy reduces the number of I/O calls thus reduces overhead and gives the MPI-IO library a possibility for further optimization. On the other side, GDDT uses an I/O library for reading data from the parallel file. Here, the data type describes the linear multidimensional section that the application requests to read. The conversion from Fortran to C array layout is done on the fly. Data for post-mortem distribution visualization is collected by means of code instrumentation. During compilation the compiler inserts calls to an instrumentation library, and produces auxiliary files containing descriptions of data (re)distribution events[2]. Each event is described by its ID, type (array distribution or redistribution), and its position in the source code. The event ID is used to find the current location of the event in the source code and the name of the array. The instrumentation subroutine converts the information about an array (i.e., type of elements, rank, shape and type of distribution) taken from the run-time array descriptors, to a textual, HPF-like format and writes it to a trace file.

[2]Here an event is understood as any change of data distribution.

4 Conclusions

In this paper we presented the advanced debugging system SPiDER integrated with the sophisticated visualization tool GDDT, which provides novel features for debugging and tuning large-scale, parallel applications: visualization of array distributions, array values, and parallel files, insight into redistributions performed during the program run by providing a replay of dynamic distributions, and advanced means for steering data distributions in a parallel program. Our work is presented in the context of HPF and the compilation environment VFC [2] (Vienna Fortran Compiler) developed at the University of Vienna, which supports HPF-2 along with a number of novel features for advanced irregular applications. However, the design concepts presented can be used in the development of programming environments for any high performance language based on the same programming model. In the next step we will develop an interface between the debugger, visualization tools and the running program, built into the runtime system, that allows to extract information describing the conceptual state of the program. This will allow to fully address all issues related to safe data redistribution.

References

[1] A Sun Microsystems, Inc. Business. *Prism 5.0 User's Guide*, July 1997.

[2] S. Benkner, K. Sanjari, V. Sipkova, and Velkov B. Parallelizing Irregular Applications with the Vienna HPF+ Compiler VFC. In *Proc. Int. Conf. High Performance Computing and Networking*, volume 1401 of *Lecture Notes in Computer Science*, pages 816–827, Amsterdam, The Netherlands, April 1998. Springer.

[3] P. Brezany, S. Grabner, K. Sowa, and R. Wismüller. DeHiFo - An Advanced HPF Debugging System. In *Proc. of the 7th Euromicro Workshop on Parallel and Distributed Processing*, pages 226–232, Madeira, Portugal, February 1999.

[4] M. Bubak, W. Funika, G. Młynarczyk, K. Sowa, and R. Wismüller. Symbol Table Management in an HPF Debugger. In *Proc. of the 7th International Conference, HPCN Europe 1999*, pages 1278–1281, Amsterdam, The Netherlands, April 1999.

[5] C. Clémençon, J. Fritscher, and R. Rühl. Visualization, Execution Control and Replay of Massively Parallel Programs within Annai's Debugging Tool. Technical Report TR-94-11, Swiss Center for Scientific Computing, 1994.

[6] Steven T. Hackstadt and Allen D. Malony. DAQV: Distributed Array Query and Visualization Framework. *Special issue on Parallel Computing*, 196(1-2):289–317, 1998.

[7] R. Koppler, S. Grabner, and J. Volkert. Visualization of Distributed Data Structures for HPF-like Languages. *Scientific Programming, spec. issue High Performance Fortran Comes of Age*, 6(1):115–126, 1997.

[8] Christian Lefebvre and Jean-Luc Dekeyser. Visualisation of HPF data mappings and of their communication cost. In *Proc. VECPAR'98*, Porto, Portugal, June 1998.