Coupling Parallel Data and Work Partitioners to the Vienna Fortran Compilation System

Peter Brezany, Viera Sipkova

Institute for Software Technology and Parallel Systems
University of Vienna, Liechtensteinstrasse 22, A-1092 Vienna, Austria
E-mail: {brezany,sipka}@par.univie.ac.at

A significant amount of software research for developing programming environments for distri-buted–memory systems is currently underway in both academia as well as industry. This paper addresses the automatic parallelization of irregular codes for distributed–memory systems. We present the language constructs provided by the Vienna Fortran to deal efficiently with irregular codes. They have been implemented within the Vienna Fortran Compilation System. This system also provides an interactive interface which enables the user to direct the system to derive data and work distributions automatically according to a selected strategy. The description of these advanced features is the main concern of the paper.

Key words: data–parallel languages, irregular computations, irregular data distribution, data–parallel loop, automatic parallelization, partitioner, inspector–executor

1 Introduction

Languages like Vienna Fortran [8], Fortran D/90D [4], or High Performance Fortran (HPF) [5], that are based on the data parallel programming model, represent a modern approach to programming distributed–memory systems. Here, programs are written using constructs of a sequential language to specify the computation on the data, and annotation constructs to specify how large data arrays and computation should be distributed among processors. The computation code is written using a global name space. The compiler analyzes the distribution annotations and generates parallel code based on the SPMD (Single–Program–Multiple–Data) programming model inserting communication statements where required by the computation.

This paper addresses the specification and automatic parallelization of advanced irregular applications for distributed–memory systems which is a challenging problem of great importance. An application is considered to be irregular if there is limited compile time knowledge about data access patterns and/or data distributions. Therefore, the process of transforming irregular codes combines compile time parallelization techniques with runtime analysis. Application areas in which such codes are found range from unstructured multigrid computation fluid dynamic solvers, through molecular dynamics codes, to time dependent flame modeling codes. In Section 2, we present the language constructs provided by Vienna Fortran for handling irregular codes. This Section also briefly outlines the compilation strategy developed and implemented for these constructs within the Vienna Fortran Compilation System (VFCS) [9]. In extensive irregular codes, using naive block distributions may result in a very inefficient implementation. VFCS provides a higher level language support and an interactive interface which enables the user to direct the system to derive irregular data and work distributions automatically according to the selected partitioning strategy. This is described in Section 3. Performance results achieved for an Euler solver kernel on the Intel iPSC/860 system are introduced in Section 4. The rest of the paper deals with related work, followed by the conclusion (Sections 5 and 6).

2 Irregular Distributions and FORALL Loops

Vienna Fortran provides regular distribution intrinsic functions BLOCK, BLOCK(M), GENERAL_BLOCK(*block-length-spec*) and CYCLIC(L). To support irregular applications, Vienna Fortran provides indirect distribution defined by means of the intrinsic function INDIRECT and a mapping array which stores a processor reference for each array element. An array whose distribution may be modified during runtime is called a *dynamically* distributed array and must be declared with the DYNAMIC attribute; else it is called a *statically* distributed array.

Irregular computations can be specified by explicit data parallel loops – forall loops. The general form of the forall loop is the following:

$$\textbf{FORALL}\ (I_1 = \boldsymbol{sec_1}, \ldots, I_n = \boldsymbol{sec_n})[\,Work\_distr\_spec \mid Partitioner\_spec]$$
forall–body
END FORALL

where for $1 \leq j \leq n$, I_j are index names, and $\boldsymbol{sec_j}$ are subscript triplets of the form (l_j, u_j, s_j). The range of the index variable I_j is specified by the corresponding section $\boldsymbol{sec_j}$. Optional *Work_distr_spec* denotes a work distribution specification which determines how loop iterations are assigned to processors for execution. *Partitioner_spec* denotes a specification of the strategy to be automatically applied by the programming environment to the forall loop for calculating optimal data and work distributions. *Work_distr_spec* and *Parti-*

```
parameter :: nnode = ..., nedge = ..., np = ...
processors pp1(np)
real , dynamic :: flux(nnode), w1(nnode), w2(nnode), w3(nnode),          &
                  w4(nnode), cc(nnode)
integer , distributed ( block ) :: map(nnode)
integer , distributed ( block , :) :: nde(nedge,2)
real :: econs, a1, a2, qs
...
... initialization of map ...
distribute flux, w1, w2, w3, w4, cc :: ( indirect (map))
...
forall i = 1, nedge
   a1 = econs * (w2(nde(i,1))+w3(nde(i,1))+w4(nde(i,1))) / w1(nde(i,1))
   a2 = econs * (w2(nde(i,2))+w3(nde(i,2))+w4(nde(i,2))) / w1(nde(i,2))
   qs = (a1 + a2) / 2.0
   reduce ( add , flux(nde(i,1)), abs(qs) + cc(nde(i,1))*econs)
   reduce ( add , flux(nde(i,2)), abs(qs) + cc(nde(i,2))*econs)
end forall
```

Fig. 1. Kernel loop of the unstructured Euler EUL3D solver

tioner_spec can be either introduced in the source program or specified interactively during the parallelization process.

Vienna Fortran provides the REDUCE statement to allow operations (e.g. global reductions) across the iterations of a parallel loop. The effect of the statement is to accumulate the values of the expression onto the target variable.

An example of Vienna Fortran code is shown in Fig.1. This code fragment represents the kernel loop from the unstructured 3–D Euler solver and will be used as a running example to illustrate our parallelization methods. The loop represents the sweep over the edges, the computationally most intensive part. The indirection array *nde* has got a block distribution and arrays *w1, w2, w3, w4, cc*, and *flux* are distributed irregularly using the indirect distribution function with the mapping array *map* which is block distributed. The values of *map* may be defined either through reading an external data file that was constructed separately from the user program, or they can be calculated dynamically at runtime employing a parallel partitioner (see Section 3).

Compilation Method

VFCS is a source–to–source translator from Vienna Fortran programs to optimized Fortran message passing programs.

In VFCS, the parallelization of irregular codes combines compile–time parallelization techniques with runtime analysis. A processing strategy based on the inspector–executor paradigm is used to implement runtime analysis and parallel execution. Each forall loop is transformed into four code phases called the *constructor of data distribution descriptors (CDDD)*, the *work distributor*, the *inspector*, and the *executor*. CDDD constructs a runtime distribution descriptor for each irregularly distributed array. The work distributor determines how to spread the iterations among the available processors. The inspector performs runtime analysis of the forall loop, generates the description of the communication necessary for the loop, and derives translation functions between the global and local accesses. The last phase, executor, performs the actual communication and executes the loop body. All those phases are supported by the runtime library that is based on the CHAOS library [7].

3 Automatic Data and Work Partitioning

Over the years researchers have developed a variety of partitioning algorithms to obtain reasonable mappings for irregular problems [6], but it is particularly troublesome to couple them to a parallel program. A *partitioning algorithm* (referred also to as *partitioner*) will decompose a dataset to a number of partitions and allocate each partition to a processor. The produced partitions need to have the minimum number of connections to other partitions (minimum communication requirements) and they should have the approximately equal number of data items (good load balancing). Partitioners typically make use of one or more of the following types of information: description of graph connectivity, spatial location of array elements, and information associating array elements with computational load. The distribution produced by these methods typically results in a table that stores a processor reference for each array element.

For each forall loop that was specified as a candidate for automatic partitioning, VFCS generates a partitioning phase including automatic determination data and work distribution before the inspector–executor phases. The essence of the implementation is contained in the CHAOS runtime library providing a number of parallel partitioners which are called from the compiler embedded code. The runtime support also includes procedures for generating interface data structures for partitioners.

Specification of the Partitioning Strategy

Currently, the CHAOS parallel partitioners implement the following partitioning methods: *Recursive Spectral Bisection (RSB)*, *Recursive Coordinate Bi-*

section (RCB), *Weighted Recursive Coordinate Bisection (WRCB)*, *Recursive Inertial Bisection (RIB)*, and *Weighted Recursive Inertial Bisection (WRIB)*. Each partitioner needs specific information and heuristics for its partitioning algorithm. The RSB partitioner decomposes data using connectivity information included in the distributed runtime data structure, called *Runtime Data Graph* (RDG). It is constructed at runtime from array access patterns occurring in the forall loop. The four other partitioners decompose data based on spatial and/or computational load information stored in data arrays.

VFCS provides the user with the opportunity to specify the appropriate partitioners and their arguments for a set of selected loops interactively. The system immediately checks the correctness of the specification, stores the specification in the internal representation of the program, and displays it at the forall loop header.

The snapshots in Figure 2 illustrate a selection of the partitioner. To specify a partitioner the user has to mark the target forall loop and select the menu [Parallelization : Partitioner]. The system then provides a list of partitioners currently available in the environment (see the upper snapshot). Then the user clicks, for example, on the [Coordinate] item to select the RCB partitioner. The system asks to pass the information about dimensionality and spatial location of array elements. The user introduces arrays (for example, X, Z and Z) as objects storing the coordinates (see the lower snapshot).

Implementation Issues

The partitioning process involves three major steps:

Data Distribution. This phase calculates how data arrays are to be decomposed by making use of the selected partitioner. In the case of the RSB partitioner, the compiler generates the coupling code which produces a RDG structure at runtime. RDG is then passed as input to the RSB partitioner. RCB, WRCB, RIB and WRIB partitioners are passed the spatial location of the array elements and/or information associating the array elements with computational load in the form of data arrays. The output of the partitioners is the new data distribution descriptor which is stored as a CHAOS construct called *translation table* representing a part of the distribution descriptor.

Work Distribution. Once the new data distribution is obtained, the loop iterations must be distributed among the processors to minimize non–local array references. For work partitioning the *almost owner computes* paradigm is chosen that assigns a loop iteration to a processor which is the owner of the largest number of distributed array elements referenced in that iteration. The work partitioner operates on the runtime data structure, called *Runtime*

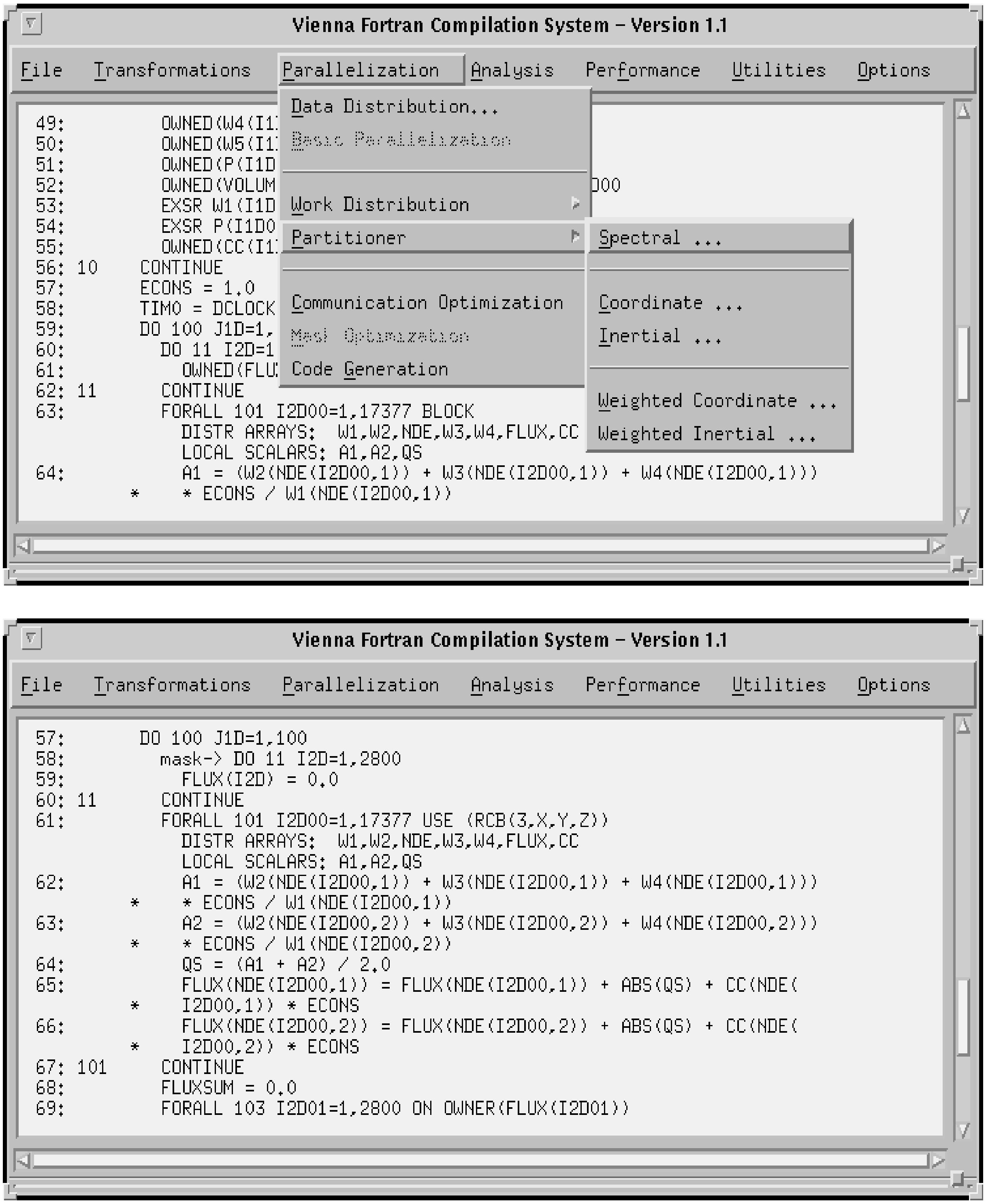

Fig. 2. Specifying a Partitioner

Iteration Graph (RIG), which lists for each iteration i all indices of each distributed array accessed in this iteration. RIG is constructed at runtime and passed as input to the work partitioner. The output of the work partitioner is a new work distribution descriptor stored as *translation table*.

Redistribution of Data and Iterations. To complete the partitioning process, all data arrays associated with the original block distribution are redistributed to the new data distribution. Similarly, the loop iterations are redistributed according to the new work distribution specification. The indirection arrays (data references) must be redistributed following the work distribution to conform with the new execution set.

A detailed description of the partitioning code for the EUL3D kernel loop given in Figure 1 can be found in [1].

4 Preliminary Performance Results

This section presents the performance of the automatically parallelized 3–D Euler solver (EUL3D), whose kernel is introduced in Section 2. We examined the code included in the outer loop with 100 iteration steps, using the input datasets with nnode = 2800 and nedge = 17377. Two versions of the code have been elaborated to evaluate the influence of different kinds of data distributions on parallel–program execution time. In the first version, all data arrays are block distributed, in the second version, data arrays accessed in irregular code parts are initially declared as block distributed, and after applying a partitioner they are redistributed irregularly according to the calculated new mapping. For lack of space we introduce only times for RSB and RCB partitioners. Times for other partitioners (RIB, WRCB and WRIB) are like the times for the RCB partitioner. Experiments were performed on Intel iPSC/860 system, the sequential code on 1 processor, the parallel code on 4, 8, and 16 processors. The generated parallel code was optimized in such a way that the partitioning code and the invariant part of the inspector was moved out of the outer loop, since the communication patterns do not change between the solver iterations. The timing results (in seconds) are summarized in Table 1.

5 Related Work

Chapman et al. [3] propose the *use–clause* for the HPF INDEPENDENT loop which enables the programmer to specify a partitioner. Ponnusamy et al. [6] propose a directive that can direct a compiler to generate a data structure on which data partitioning is to be based and a directive for linking this data structure to a partitioner. An experimental Fortran 90D implementation is able to couple the CHAOS partioners to simple forall statements. In [2], we describe the utilization of an external partitioner.

Table 1
Performance results of the EUL3D solver

Sequential code									
35.37									
Parallel code									
Comput. Phases	Irregular Distribution						Block Distribution		
	RSB			RCB					
	Number of Processors								
	4	8	16	4	8	16	4	8	16
Data Part.	3.20	10.92	12.47	0.17	0.18	0.33	–	–	–
Work Part.	0.59	0.40	0.30	0.59	0.35	0.25	–	–	–
Total	20.31	19.56	17.14	15.64	8.23	4.91	25.84	18.06	13.14

6 Conclusion

In this paper we have introduced a progressive technology for developing advance irregular applications for massively parallel systems. The work has been inspired by the CHAOS project conducted at the University of Maryland. In VFCS, the techniques supporting automatic work and data partitioning are fully integrated with the compile time techniques. The implementation is currently in a stage of testing and benchmarking on real codes.

Acknowledgement

The authors would like to thank Joel Saltz and his research group for providing the PARTI and CHAOS libraries, and for their valuable help by the implementation of the VFCS–CHAOS interface.

References

[1] P. Brezany, B. Chapman, R. Ponnusamy, V.Sipkova, H.P.Zima, *Study of Application Algorithms with Irregular Distributions* (Report D1Z-3 of the CEI-PACT Project, University of Vienna, April 1994).

[2] P. Brezany, V. Sipkova, B. Chapman, R. Greimel, *Automatic Parallelization of the AVL FIRE Benchmark for a Distributed-Memory System*, (Proc. of PARA95 Workshop, Denmark, August 1995).

[3] B. Chapman, P. Mehrotra, H. Zima, *Extending HPF for Advanced Data Parallel Application* (Tech. Report TR 94-7, University of Vienna, May 1994).

[4] G. Fox, S. Hiranandani, K. Kennedy, *Fortran D Language Specification* (Tech. Report COMP TR90-141, Rice University, December 1990).

[5] High Performance Fortran Forum, *High Performance Fortran Language Specification* (Tech. Report, Rice University Houston Texas, May 1993).

[6] R. Ponnusamy, J. Saltz, A. Choudhary, Y.-S. Hwang, G. Fox, *Runtime–Compilation Techniques for Data Partitioning and Communication Schedule Reuse* (CS-TR-93-32, University of Maryland, College Park, MD, April 1993).

[7] J. Saltz et al., *A Manual for the CHAOS Runtime Library* (Tech. Report, University of Maryland, College Park, MD 20742, May 1994).

[8] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, A. Schwald, *Vienna Fortran - A language Specification V 1.1* (ACPC-TR 92-4, Univ. of Vienna, March 1992).

[9] H. Zima et al., *Vienna Fortran Compilation System* (User's Guide, University of Vienna, October 1995).