Automatic Parallelization of the AVL FIRE Benchmark for a Distributed-Memory System*

Peter Brezany[a], Viera Sipkova[a], Barbara Chapman[b], Robert Greimel[c]

[a]Institute for Software Technology and Parallel Systems
[b]European Centre for Parallel Computing at Vienna
University of Vienna, Liechtensteinstrasse 22, A-1092 Vienna, Austria
[c]AVL List GmbH, Kleiststrasse 48, A-8020 Graz, Austria

To appear in Proc. of the PARA95 Workshop, Copenhagen, Denmark

Abstract. Computational fluid dynamics (CFD) is a Grand Challenge discipline whose typical application areas, like aerospace and automotive engineering, often require enormous amount of computations. Parallel processing offers very high performance potential, but irregular problems like CFD have proven difficult to map onto parallel machines. In such codes, access patterns to major data arrays are dependent on some runtime data, therefore runtime preprocessing must be applied on critical code segments. So, automatic parallelization of irregular codes is a challenging problem. In this paper we describe parallelizing techniques we have developed for processing irregular codes that include irregularly distributed data structures. These techniques have been fully implemented within the Vienna Fortran Compilation System. We have examined the AVL FIRE benchmark solver GCCG, to evaluate the influence of different kinds of data distributions on parallel-program execution time. Experiments were performed using the Tjunc dataset on the iPSC/860.

1 Introduction

Computational Fluid Dynamics (CFD) is a Grand Challenge discipline which has been applied as a successful modelling tool in such areas as aerospace and automotive design. CFD greatly benefits from the advent of massively parallel supercomputers [12]. On the close cooperation between researchers in CFD and specialists in parallel computation and parallel programming, successful CFD modelling tools were developed. This paper deals with the automatic parallelization of the FIRE solver benchmark developed at AVL Graz, Austria.
The CFD software **FIRE** is a general purpose computational fluid dynamics program package. It was developed specially for computing compressible and incompressible turbulent fluid flows as encountered in engineering environments.

* The work described in this paper was carried out as part of the European ESPRIT project PPPE and CEI-PACT project funded by the Austrian Science Foundation (FWF) and the Austrian Ministry for Science and Research (BMWF).
Authors' e-mail addresses: {brezany,barbara,sipka}@par.univie.ac.at, rg@avl.co.at

Two- or three-dimensional unsteady simulations of flow and heat transfer within arbitrarily complex geometries with moving or fixed boundaries can be performed.
For the discretization of the computational domain a finite volume approach is applied. The resulting system of strongly coupled nonlinear equations has to be solved iteratively. The solution process consists of an outer non-linear cycle and an inner linear cycle. The matrices which have to be solved in the linear cycle are extremely sparse and have a large and strongly varying bandwidth. In order to save memory, only the non-zero matrix elements are stored in linear arrays and are accessed by indirect addressing.
Automatic parallelization of irregular codes like FIRE is a challenging problem. In irregular codes, the array accesses cannot be analyzed at compile time to determine either independence of these or to find what data must be pre-fetched and where it is located. Therefore, the appropriate language support is needed, as well as compile time techniques relying on runtime mechanisms. The **Vienna Fortran** (VF) language [14] provides several constructs to deal efficiently with irregular codes. These include constructs for specifying irregular distributions and for explicitly specifying asynchronous parallel loops (FORALL loops).

We have transformed one benchmark solver (**GCCG**, orthomin with diagonal scaling [1]) of the AVL FIRE package to VF and parallelized it by the **Vienna Fortran Compilation System** (VFCS). Because, access patterns to data arrays of the inner cycle are not known until runtime, a parallelization method based on the combination of compile time and runtime techniques has been applied.
In Section 2 the method and data structures of the GCCG program are discussed. Section 3 describes the data structures as they are specified by VF. We focus on the selection of the appropriate data and work distribution for irregular code parts. We elaborated two versions of the program. In the first version, the INDIRECT data distribution was specified for the data arrays accessed in the irregular code parts, and in the second one, all arrays got BLOCK data distributions. The mapping array used for the irregular distribution was determined by an external partitioner. Section 3 also describes the automatic parallelization strategy which the VFCS applies to irregular codes and the interface to runtime support. Performance results for the Tjunc dataset achieved for both GCCG program versions on the Intel iPSC/860 system are discussed in Section 4. The rest of the paper deals with related work (Section 5), followed by the conclusion (Section 6). Concepts described in this paper can be used in the compiler of any language that is based on the same programming model as VF.

2 Survey of the GCCG solver

The FIRE benchmark consists of solving the linearized continuity equation with selected input datasets using the unstructured mesh linear equation solver GCCG. Each dataset contains coefficients, sources, and the addressing array including linkage information. The numerical solution to differential equation describing the transport of a scalar variable Φ is performed with the finite volume method. For the discrete approximation the computational domain is subdivided into a finite number of hexahedral elements, the control volumes or *internal* cells. A compass notation is used to identify the interconnections between the centres of the control volumes (E=East, W=West, etc.). Integrating over the control volumes results in a system of non-linear algebraic equations:

$$A_p \; \Phi_p = \sum_{c=\{E,S,N,W,SE,\ldots\}} A_c \Phi_c + S_\Phi \tag{1}$$

where the pole coefficient A_p can always be written as the sum of the neighbour coefficients in different directions. Due to the locality of the discretisation method equation (1) represents an extremely sparse system of algebraic equations which has to be solved iteratively. The solution process consists of an outer cycle dealing with the non-linearities and an inner cycle dealing with the solution to the linearized equation systems. In particular, the outer cycle is designed to update the coefficients and sources from the previous iterations in order to achieve strong coupling between momentum and pressure. The innermost cycle consists of solving the linear equation systems:

$$A \; \Phi = S_\Phi \tag{2}$$

for every flow variable Φ. The main diagonal of A consists of the pole coefficient A_p, and the sidebands of A are obtained from the neighbouring node coefficients A_c of (1). Both, the coefficients and the sources are kept constant during the iterative solution steps. To solve the linear equation system the truncated Krylov subspace method Orthomin is used [13].
The solution process stops when the ratio of the residual vector R^n at iteration n and the residual vector R^0 at iteration 0 falls below a small value ϵ :

$$\| \; R^n \; \| = \| \sum_{c=\{E,S,\ldots\}} A_c \Phi_c^n + S_\Phi - A_p \Phi_p^n \; \| < \; \epsilon \cdot \| \; R^0 \; \| \tag{3}$$

For boundary conditions an additional layer of infinitely thin cells is introduced around the computational domain, called *external* cells.
Flow variables, coefficients and sources associated with each cell are held in 1-dimensional arrays of two different sizes: one size corresponds to the number of internal cells (parameter NNINTC), and the second size corresponds to the total number of cells, internal plus external cells (parameter NNCELL). These two kinds of arrays are related with each other in such a way that the first portion of the bigger arrays (including internal cells), is aligned with the smaller arrays. To determine the interconnection of cells (links cell-centre to cell-centre) an indirect addressing scheme is used whereby a unique number, the *address*, is associated with each cell. These linkage information is stored in the 2-dimensional array LCC, where the first index stands for the actual cell number and the second index denotes the direction to its neighbouring cell. The value of LCC(cell number, direction) is the cell number of the neighbouring cell. All calculations in

the solver GCCG are carried out over the internal cells. External cells can only be accessed from the internal cells, no linkage exists between the external cells.

3 Automatic Parallelization of GCCG

Initially, the sequential code of the GCCG solver has been ported to the VF code. Declarations of arrays with distribution specifications and the main loop within the outer iteration cycle is outlined in Figure 1. This code fragment will be used as a running example to illustrate our parallelization method.

```
    PARAMETER :: NP = ...
    PARAMETER :: NNINTC = 13845, NNCELL = 19061
    PROCESSORS P1D(NP)
    DOUBLE PRECISION, DIMENSION(NNINTC),                      &
       DISTRIBUTED(BLOCK) :: BP, BS, BW, BL, BN, BE, BH,      &
                             DIREC2, RESVEC, CGUP
    INTEGER, DISTRIBUTED(BLOCK, :) :: LCC(NNINTC, 6)
S1  DOUBLE PRECISION, DIMENSION(NNCELL), DYNAMIC ::           &
                             DIREC1, VAR, DXOR1, DXOR2
S2  INTEGER, DISTRIBUTED(BLOCK) :: MAP(NNCELL)
    ...
S3  READ ( u) MAP
S4  DISTRIBUTE DIREC1, VAR, DXOR1, DXOR2 :: INDIRECT (MAP)
    ...
F1  FORALL nc = 1, NNINTC  ON OWNER(DIREC2(nc))
          DIREC2(nc) =   BP(nc) * DIREC1(nc)              &
                       - BS(nc) * DIREC1(LCC(nc, 1))      &
                       - BW(nc) * DIREC1(LCC(nc, 4))      &
                       - BL(nc) * DIREC1(LCC(nc, 5))      &
                       - BN(nc) * DIREC1(LCC(nc, 3))      &
                       - BE(nc) * DIREC1(LCC(nc, 2))      &
                       - BH(nc) * DIREC1(LCC(nc, 6))
    END FORALL
    ...
```

Fig. 1. Kernel loop of the GCCG solver

Arrays DIREC1, VAR, DXOR1 and DXOR2 are distributed irregularly using the **INDIRECT** distribution function with the mapping array MAP which is initialized from a file referred to as mapping file; the remaining arrays have got the BLOCK distribution.
The mapping file has been constructed separately from the program through the

Domain Decomposition Tool (DDT) [8] employing the recursive spectral bisection partitioner. The DDT inputs the computational grid corresponding to the set of internal cells (see Figure 2), decomposes it into a specified number of partitions, and allocate each partition to a processor. The resulting mapping array is then extended for external cells in such a way that the referenced external cells are allocated to that processor on which they are used, and the non-referenced external cells are spread in a round robin fashion across the processors. Arrays VAR, DXOR1 and DXOR2 link DIREC1 to the code parts that don't occur in Figure 1.

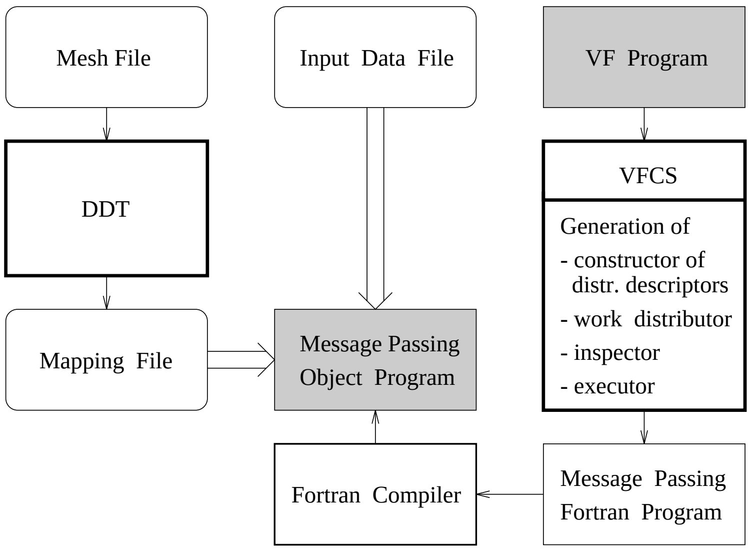

Fig. 2. Flow of Data in Compilation and Execution Phases

The code version that only includes regular distributions is simply derived from Figure 1 by replacing the keyword **DYNAMIC** in statement S1 by the distribution specification: **DISTRIBUTED**(**BLOCK**), and removing statements S2, S3 and S4. The main loop (F1) within the outer iteration cycle computes a new value for every cell by using the old value of this cell and of its six indirectly addressed neighbored cells. This loop has no dependences and can be directly transformed to the parallel loop. Work distribution **ON OWNER**(DIREC2(nc)) ensures that communications are caused only by the array DIREC1. The remaining loops, with exception of the one that updates the DIREC1, perform either calculations on local data only or global operations (global sums).
In the rest of this section, we first describe the parallelization strategy applied to the code version from Figure 1 and then briefly discuss simplifications applied when processing the code containing regularly distributed arrays only. Our ap-

proach is based on a model that is graphically sketched in Figure 2. The method applied by DDT is outlined on the previous page. The strategy used by the VF-CS generates four code phases for the above kernel, called the **constructor of data distribution descriptors** (CDDD), **work distributor**, **inspector**, and **executor** [10]. CDDD constructs a runtime distribution descriptor for each irregularly distributed array. The work distributor determines how to spread the work (iterations) among the available processors. The inspector analyzes the communication patterns of the loop, computes the description of the communication, and derives translation functions between global and local accesses, while the executor performs the actual communication and executes the loop iterations. All the phases are supported by the appropriate runtime library that is based on the PARTI library developed by Saltz and his coworkers [7].

```
INTEGER :: MAP(local$^{MAP}$_size)
INTEGER, DIMENSION(:), POINTER :: local$^{DIREC1}$, local$^{DIREC1}$_size
TYPE (TT_EL_TYPE), DIMENSION(:), POINTER :: tt$^{DIREC1}$
TYPE (DIST_DESC) :: dd$_{DIREC1}$
    ...
 C--Constructing local index set for DIREC1 using the MAP values
      CALL build_Local(dd$_{MAP}$, MAP, local$^{DIREC1}$, local$^{DIREC1}$_size)
 C--Constructing translation table for DIREC1
      tt$^{DIREC1}$ = build_TRAT(local$^{DIREC1}$_size, local$^{DIREC1}$)
 C--Constructing runtime distribution descriptor for DIREC1
      $dd_{DIREC1}$%local = local$^{DIREC1}$
      $dd_{DIREC1}$%local_size = local$^{DIREC1}$_size
      $dd_{DIREC1}$%tt = tt$^{DIREC1}$
      ... initialization of other fields of $dd_{DIREC1}$ ...
```

Fig. 3. CDDD for array DIREC1

3.1 Constructor of Data Distribution Descriptors (CDDD)

For each irregularly distributed array A and in each processor p, CDDD constructs the runtime distribution descriptor $dd_A(p)$ which includes information about: shape, alignment and distribution, associated processor array, and size of the local data segment of A in processor p. In particular, it includes the local index set $local^A(p)$ and the translation table tt^A. The set $local^A(p)$ is an ordered set of indices designating those elements of A that have to be allocated in a particular processor p. The translation table tt^A is a distributed data structure which records the home processor and the local address in the home processor's memory for each array element of A. Only one distribution descriptor is constructed for a set of arrays having the same distribution. The construction of

dd_{DIREC1} (it also describes the distribution of VAR, $DXOR1$ and $DXOR2$) is illustrated by Figure 3. On each processor p, the procedure *build_Local* computes the set $local^{DIREC1}(p)$ and its cardinality denoted by variables $local^{DIREC1}$ and $local^{DIREC1}$*_size* respectively. These two objects are input arguments to the PARTI function *build_TRAT* that constructs the translation table for *DIREC1* and returns a pointer to it.

3.2 Work Distributor

On each processor p, the work distributor computes the *execution set exec(p)*, i.e. the set of loop iterations to be executed on processor p.
For the GCCG kernel loop in Figure 1, the execution set can be determined by a simple set operation (see [2]):

$$exec(p) = [1 : NNINTC] \cap local^{DIREC2}(p)$$

VF provides a wide spectrum of possibilities for the work distribution specification. The appropriate techniques for processing individual modes are described in [3, 4, 5, 6].

3.3 Inspector, Executor, and PARTI support

The **inspector** performs a dynamic loop analysis. Its task is to describe the necessary communication by a set of so called *schedules* that control runtime procedures moving data among processors in the subsequent executor phase. The dynamic loop analysis also establishes an appropriate addressing scheme to access local elements and copies of non-local elements on each processor. The inspector generated for the GCCG kernel loop is introduced in Figure 4.
Information needed to generate a schedule, to allocate a communication buffer, and for the global to local index conversion for the rhs array references can be produced from the appropriare *global reference* lists, along with the knowledge of the array distributions. Each global reference list $globref_i$ is computed from the subscript functions and from *exec(p)*[2] (see lines G1–G16 of Figure 4). The list $globref_i(p)$, its size and the distribution descriptor of the referenced array are used by the Parti procedure localize to determine the appropriate *schedule*, the size of the communication buffer to be allocated, and the local reference list $locref_i(p)$ which contains results of global to local index conversion. The declarations of rhs arrays in the message passing code allocate memory for the local segments holding the local data and the communication buffers storing copies of non-local data. The buffers are appended to the local segments. An element from $locref_i(p)$ refers either to the local segment of the array or to the buffer. The procedure ind_conv performs the global to local index conversion for array references that don't refer non-local elements.
The **executor** is the final phase in the execution of the FORALL loop; it performs communication described by schedules, and executes the actual computations for all iterations in *exec(p)*. The schedules control communication in such a

[2] In Figure 4, *exec(p)* and its cardinality are denoted by the variables *exec* and *exec_size* respectively.

```
 C--INSPECTOR code

 C--Constructing global referece list for the 1st dimension of LCC
G1   n1 = 1
G2   DO k=1, exec_size
G3      globref1(n1) = exec(k); n1 = n1 + 1
G4   END DO
 C--Index Conversion for LCC having the data distribution desc. dd_1
     CALL  ind_conv (dd_1, globref1, locref1, n1-1)
 C--Constructing global reference list for DIREC1 and
 C--DIREC2, BP, BS, BW, BL, BN, BE, BH
G5   n1 = 1; n2 = 1; n3 = 1
G6   DO k=1, exec_size
G7      globref2(n2) = exec(k); n2 = n2 + 1
G8      globref3(n3) = exec(k)
G9      globref3(n3+1) = LCC(locref1(n1),1)
G10     globref3(n3+2) = LCC(locref1(n1),4)
G11     globref3(n3+3) = LCC(locref1(n1),5)
G12     globref3(n3+4) = LCC(locref1(n1),3)
G13     globref3(n3+5) = LCC(locref1(n1),2)
G14     globref3(n3+6) = LCC(locref1(n1),6)
G15     n3 = n3 + 7; n1 = n1 + 1
G16  END DO
 C--Index Conversion for DIREC2, BP, BS, BW, BL, BN, BE, BH
 C--having the common data distribution descriptor dd_2
     CALL  ind_conv (dd_2, globref2, locref2, n2-1)
 C--Computing schedule and local reference list for DIREC1
     CALL  localize(tt^DIREC1, sched3, globref3, locref3, n3-1, nonloc3)

 C--EXECUTOR code

 C--Gather non-local elements of DIREC1
     CALL  gather(DIREC1(1), DIREC1(local^DIREC1_size+1), sched3)
 C--Transformed forall loop
     n2 = 1; n3 = 1
     DO k=1, exec_size
        DIREC2(locref2(n2)) = BP(locref2(n2)) * DIREC1(locref3(n3))      &
             - BS(locref2(n2)) * DIREC1(locref3(n3+1))      &
             - BW(locref2(n2)) * DIREC1(locref3(n3+2))      &
             - BL(locref2(n2)) * DIREC1(locref3(n3+3))      &
             - BN(locref2(n2)) * DIREC1(locref3(n3+4))      &
             - BE(locref2(n2)) * DIREC1(locref3(n3+5))      &
             - BH(locref2(n2)) * DIREC1(locref3(n3+6))
        n2 = n2 + 1; n3 = n3 + 7
     END DO
```

Fig. 4. Inspector and Executor for the GCCG kernel loop

way that execution of the loop using the local reference list accesses the correct data in local segments and buffers. Non-local data needed for the computations on processor p are gathered from other processors by the runtime communication procedure gather[3]. It accepts a schedule, an address of the local segment, and an address of the buffer as input arguments.

4 Performance Results

This section presents the performance of the automatically parallelized benchmark solver GCCG for two types of data distribution: the regular BLOCK, and the irregular distribution according to the mapping array. The generated code was slightly optimized by hand in such a way that the inspector was moved out of the outer iteration cycle, since the communication patterns do not change between the solver iterations.

Dataset: TJUNC		
Number of internal cells: 13845		
Number of total cells: 19061		
Number of solver iterations: 338		
Sequential code		
Number of Processors	Time (in secs)	
1	43.47	
Parallel code		
Number of Processors	Time (in secs)	
	Data Distr: BLOCK	Data Distr: INDIRECT
4	30.25	27.81
8	24.66	19.50
16	22.69	13.50

Table 1. Performance results of the GCCG solver

We examined the GCCG using the input dataset Tjunc, with 13845 internal cells and 19061 total number of cells. Calculation stopped after the 338 iterations. The program has been executed on the iPSC/860 system, the sequential code on 1 processor, the parallel code on 4, 8, and 16 processors. The results are sumarized in the Table 1. Figure 5 shows the speedup defined by the ratio of the

[3] If necessary, space for the local segment and buffer is reallocated prior to the gather call, depending on the current size of the segment and the number of non-local elements computed by localize.

single processor execution time T_s to the multi-processor execution time T_{np}, where np stands for the number of processors.

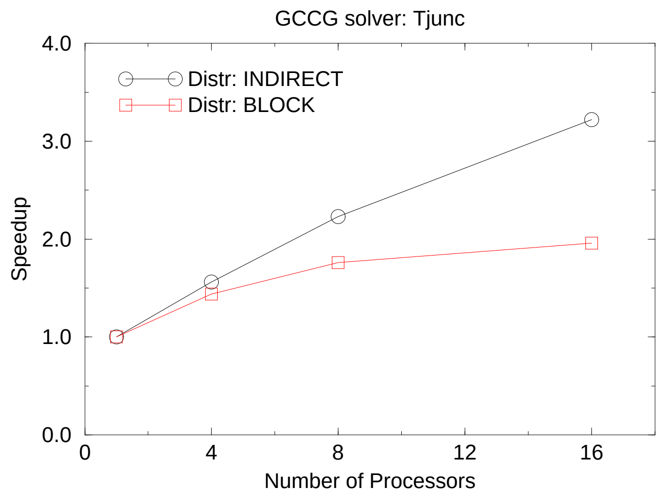

Fig. 5. Speedup of the GCCG solver

5 Related Work

Solutions of many special cases of the problems discussed in this paper appeared in the literature. Koelbel [10] introduces the term inspector/executor for processing irregular data-parallel loops. PARTI [7] has been the first runtime system constructed to support the handling of irregular computations on massively parallel systems on the basis of the inspector/executor paradigm. Van Hanxleden [9] developes a method for optimizing communication placement in irregular codes. Techniques developed for automatic coupling parallel data and work partitioners are described in [6, 11].

6 Conclusions

Our experiments have shown that in irregular codes like the AVL FIRE Benchmark it is advantageous to use irregular distributions. There are many heuristic methods to obtain irregular data distribution based on a variety of criteria; different partitioners that have been developed in the last years offer the implementation of those methods. VFCS automatically generates interface to external and on-line partitioners on the basis of the information provided by the user and derived by the compile time program analysis. This paper described the automatic parallelization method based on the utilization of the external partitioner.

Acknowledgment

The authors would like to thank Edi Mehofer and Bernd Wender for their helpful discussions during the design of the VFCS support for irregular distributions.

References

1. G. Bachler, R. Greimel. *Parallel CFD in the Industrial Environment.* Unicom Seminars, London, 1994.
2. S. Benkner, P. Brezany, H.P. Zima. *Processing Array Statements and Procedure Interfaces in the Prepare High Performance Fortran Compiler.* Proc. 5th International Conference on Compiler Construction, Edinburgh, U.K., April 1994, Springer-Verlag, LNCS 786, pp. 324–338.
3. P. Brezany, M. Gerndt, V. Sipkova, and H.P. Zima. *SUPERB Support for Irregular Scientific Computations.* In Proceedings of the SHPCC '92, Williamsburg, USA, April 1992, pp.314-321.
4. P. Brezany, B. Chapman, R. Ponnusamy, V. Sipkova, and H Zima, *Study of Application Algorithms with Irregular Distributions,* tech. report D1Z-3 of the CEI-PACT Project, University of Vienna, April 1994.
5. P. Brezany, O. Chéron, K. Sanjari, and E. van Konijnenburg, *Processing Irregular Codes Containing Arrays with Multi-Dimensional Distributions by the PREPARE HPF Compiler,* HPCN Europe'95, Milan, Springer-Verlag, 526–531.
6. P. Brezany, V. Sipkova. *Coupling Parallel Data and Work Partitioners to VFCS.* Submitted to the Conference EUROSIM – HPCN Challenges 1996, Delft.
7. R. Das, and J. Saltz. *A manual for PARTI runtime primitives - Revision 2.* Internal Research Report, University of Maryland, Dec. 1992.
8. N. Floros, J. Reeve. *Domain Decomposition Tool (DDT).* Esprit CAMAS 6756 Report, University of Southampton, March 1994.
9. R. van Hanxleden. *Compiler Support for Machine-Independent Parallelization of Irregular Problems.* Dr. Thesis, Center for Research on Parallel Computation, Rice University, December 1994.
10. C. Koelbel. *Compiling Programs for Nonshared Memory Machines.* Ph.D. Dissertation, Purdue University, West Lafayette, IN, November 1990.
11. R. Ponnusamy, J. Saltz, A. Choudhary, Y-S. Hwang, G. Fox. *Runtime Support and Compilation Methods for User-Specified Data Distributions.* Internal Research Report, University of Maryland, University of Syracuse, 1993.
12. H.D. Simon. *Parallel CFD.* The MIT Press, Cambridge, 1992.
13. P.K.W. Vinsome. *ORTHOMIN, an iterative method for solving sparse sets of simultaneous linear equations.* In Proc. Fourth Symp. on Reservoir Simulation, Society of Petroleum Engineers of AIME, pp. 149–159.
14. H. Zima, P. Brezany, B. Chapman, P. Mehrotra, A. Schwald. *Vienna Fortran - A language Specification Version 1.1.* University of Vienna, ACPC-TR 92-4, 1992.