VFC: The Vienna Fortran Compiler [1]

Siegfried Benkner [2]
Institute for Software Technology and Parallel Systems, University of Vienna, Liechtenstein Strasse 22, A-1090 Vienna, Austria

High Performance Fortran (HPF) offers an attractive high-level language interface for programming scalable parallel architectures providing the user with directives for the specification of data distribution and delegating to the compiler the task of generating an explicitly parallel program. Available HPF compilers can handle regular codes quite efficiently, but dramatic performance losses may be encountered for applications which are based on highly irregular, dynamically changing data structures and access patterns. In this paper we introduce the Vienna Fortran Compiler (VFC), a new source-to-source parallelization system for HPF+, an optimized version of HPF, which addresses the requirements of irregular applications. In addition to extended data distribution and work distribution mechanisms, HPF+ provides the user with language features for specifying certain information that decisively influence a program's performance. This comprises data locality assertions, non-local access specifications and the possibility of reusing runtime-generated communication schedules of irregular loops. Performance measurements of kernels from advanced applications demonstrate that with a high-level data parallel language such as HPF+ a performance close to hand-written message-passing programs can be achieved even for highly irregular codes.

Keywords: High Performance Fortran, data parallel programming, data distribution, parallelizing compilers, communication schedule reuse

1. Introduction

The High Performance Fortran Forum (HPFF), which first convened during 1992, set itself the task of defining language extensions for Fortran to facilitate data parallel programming on a wide range of parallel architectures, without sacrificing performance. Much of the work on HPF-1 [19] focused on extending Fortran 90 by directives for specifying at a high level of abstraction regular data distributions in the context of a single-threaded data parallel paradigm with a global address space. Other extensions allow the specification of explicitly parallel loops, in particular with the `FORALL` statement and construct and the `INDEPENDENT` directive, as well as a number of library routines.

A major advantage of the HPF approach compared to parallel programming based on message-passing is that the user writes a sequential program and specifies how the data space of a program should be distributed by adding data distribution directives to the declarations of arrays. It is then the responsibility of the compiler to translate a program containing such directives into an efficient parallel Single-Program-Multiple-Data (SPMD) target code using explicit constructs for data-sharing, such as message-passing on distributed memory machines. This style of programming hides the machine specific details in the compiler enabling the user to concentrate on the algorithmic issues. As a consequence, HPF programs are not only much simpler than message-passing programs but also much more flexible. Changing, for example, the data layout requires only minor program changes.

Although HPF-1 was very well suited for applications based on regular data structures and access patterns, it soon became apparent that it did not provide enough flexibility for an efficient formulation of many advanced algorithms. For example, weather forecasting codes [2] or crash simulations [15], are often characterized by the need to distribute data in an irregular manner and dynamically balance the computational load of the processors. The current version of HPF, HPF-2 [20], and in particular its "approved extensions", is a significant step in supporting such algorithms.

The study of real applications has, however, revealed that even with the enhanced data and work distributions offered by the latest version of the language, certain information that may decisively influence a program's performance cannot be expressed. One such example is information about the invariance of ar-

[1]The work described in this paper was partially supported by the ESPRIT IV Long Term Research Project 21033 "HPF+" of the European Commission.

[2]Present address: C&C Research Laboratories, NEC Europe Ltd., Rathausallee 10, 53757 Sankt Augustin, Germany. E-mail: benkner@ccrl-nece.technopark.gmd.de.

Scientific Programming 7 (1999) 67–81
ISSN 1058-9244 / $8.00

ray access patterns in irregular parallel loops. In such loops arrays are accessed indirectly via dynamically defined index arrays. Consequently an HPF compiler can not statically determine the access pattern and derive the required communication in the parallel target program. Therefore, most HPF compilers generate code which at runtime analyzes the indirect accesses to distributed arrays and determines a communication schedule to handle the gathering and scattering of nonlocal data required for executing the parallel loop. This pre-processing phase to compute a schedule is called *inspector* [16]. Inspectors may be very time-consuming in some cases dominating the execution time for a whole program; as a consequence, the avoidance of redundant inspector executions is a key issue when parallelizing irregular codes. However existing compile-time and runtime analysis techniques [28,30] are not powerful enough to detect all instances of redundant inspectors. Thus, language support is required for this problem.

HPF+ [3], an improved variant of HPF for advanced industrial applications which has been developed in the Research Project "HPF+", addresses these issue by providing language features to express the redundancy of schedule computations. Moreover, HPF+ offers mechanisms to influence the mapping of computations to processors, to assert the locality of computations, and to specify in addition to the distribution of an array the possibly irregular areas of non-local accesses.

In the remainder of this paper we describe the HPF+ compiler VFC with a special focus on language features and parallelization techniques for advanced irregular applications which are currently not adequately supported by the de-facto standard HPF-2 and commercial HPF compilers. In Section 2 we describe the overall structure of VFC followed by an overview of the supported HPF+ language features in Section 3. In Section 4 we present the main parallelization techniques implemented in VFC. Section 5 presents performance results of two irregular kernels developed in the "HPF+" project. In the remaining sections we discuss related and future work.

2. The VFC System

VFC is a source-to-source parallelization system that translates Fortran95/HPF+ programs to Fortran90/MPI message-passing SPMD programs. The system is based on a Fortran 90 frontend initially developed at GMD Berlin and extended at the University of Vienna. Besides the main implementation languages C and C++, the Cocktail compiler construction toolbox [18] has been used extensively.

The system is currently available for the Solaris operating system. Codes parallelized with VFC can be executed on the QSW (Meiko) CS-2, the NEC Cenju-3, the Fujitsu AP-3000, and on clusters of workstations.

VFC is the first HPF compilation system that offers besides the basic block and cyclic data distributions, new data distribution formats required for irregular codes, such as the generalized block distributions and indirect distributions, and that fully implements dynamic data redistribution. Unlike most other HPF compilers, VFC provides powerful parallelization strategies for a large class of non-perfectly nested loops with irregular runtime-dependent access patterns which are common in industrial codes. In this context, the concept of communication schedule reuse and halo areas (see Section 3), allows the user to reduce or amortize the potentially large overhead of the associated runtime compilation strategies.

2.1. System components

The VFC system consists of six major components: the frontend, the *analysis module*, the parallelization module, the instrumentation system, the code generator, and the runtime system.

The *frontend* performs scanning and parsing of the input program, and constructs an intermediate representation of the program in the form of an abstract syntax tree.

The analysis module constructs different abstractions of the program such as the program dependence graph, the call graph and a flow graph for each program unit. These data structures are the basis for data dependence analysis and for an advanced interprocedural analysis framework [24] which is currently being developed.

The *parallelization module*, which transforms the input program into a parallel message-passing SPMD program, consists of several components. The *normalization component* performs various transformations such as array and loop normalization, normalization of data distribution directives, substitution of function calls in expressions that reference distributed objects, normalization of procedure interfaces, and others. The task of the *classification component* is to determine the parallelization strategy that should be applied to each executable construct. A generalized in-

spector/executor strategy is utilized for parallelizing irregular loop nests. A parallelization technique based on intersection of regular sections is applied to regular loops and explicitly parallel constructs such as array assignments. A default strategy utilizing single-element send/receive/broadcast operations, is applied either to scalar assignments or to non-parallelizable loops. Depending on the availability of analysis information or the presence of user assertions, different variants of these strategies are applied. A major emphasis has been put on providing feedback to the user about potential overheads introduced by the compiler during parallelization. For example, if VFC has to sequentialize independent loops, if communication statements are generated within loops, or if implicit redistributions have to be generated at procedure boundaries, the user is informed by means of *diagnostic messages*.

The *instrumentation component* [10] inserts calls to a measurement library in order to generate MPI trace-files during execution of the program. The instrumentation can be triggered by the user by means of command-line options, and it supports selective program instrumentation, in order to provide the user with detailed information about the execution of the parallelized code. VFC trace-files can be analyzed by post-execution performance analysis tool such as MEDEA [10,11] and others.

The *code generator* translates the internal representation of the program into Fortran 90 source code, which is then submitted to the Fortran 90 compiler of the parallel target machine in order to generate the executable parallel program.

The runtime system of VFC consists of two major parts: the VHPF and the PARS libraries. The main task of the VHPF library is the management of distributed objects at runtime, including runtime support for dynamic data redistribution. Moreover, the VHPF library provides high level interfaces for communication primitives of the PARS library and the ADLIB library [12]. The PARS library is an extension of the PARTI [16] library to support multidimensionally distributed arrays. It provides communication routines based on MPI-1 [26] and supports all aspects of the inspector/executor runtime parallelization strategy, including the support for schedule reuse and halos. The ADLIB library is utilized for the parallelization of array assignment statements and intrinsic functions in the context of regular distributions.

2.2. Input language

The input language of VFC is Fortran 95 with HPF+ extensions. Although VFC supports full Fortran 95 in the context of non-distributed data objects, the system imposes certain restrictions on the transfer of distributed arguments at procedure boundaries, the input/output of distributed data, and the use of intrinsic functions. In particular, VFC neither supports the reshaping of distributed arrays at procedure boundaries nor the transfer of sections of distributed arrays to procedures.

The supply of important modularization features of Fortran such as procedure interfaces, internal procedures, and modules and the support for separate compilation significantly alleviates the parallelization of large programs.

HPF+ not only comprises the de-facto standard HPF-2 but also includes many features that have been adopted by the Approved Extensions of HPF-2. In addition HPF+ simplifies the distribution model of HPF without sacrificing required functionality, and introduces new features to provide the compiler with essential information on the applicability of crucial optimizations.

3. HPF+ language features

In the following we give a brief overview of the HPF+ language as supported by VFC with a special focus on features required for an efficient parallelization of advanced irregular applications. For a more detailed description of HPF+ the reader is referred to [3]. In Table 1 a comparison of the HPF+ language features with those provided in the HPF-2 core language and the Approved Extensions of HPF-2 is presented.

3.1. Processor arrays

Arbitrary multidimensional processor arrays may be defined in order to provide an abstraction from the topology of the parallel target architecture. Programs may be parallelized for an arbitrary number of processors by using the intrinsic function `NUMBER_OF_PROCESSORS()` in processor declarations. An explicit relationship between different processor arrays may be established based on the concept of *processor views* by means of the `RESHAPE` intrinsic function. This feature enables the user to impose different views on the set of available processors, to define named processor array sections, and to permute the dimensions of multidimensional processor arrays.

Table 1
A comparison of selected language features in HPF+, HPF 2 and the HPF 2 Approved Extensions

Feature	HPF+	HPF-2	A.Ext.
Processors			
processor views	+	–	–
Data distribution			
`BLOCK` distributions	+	+	+
`CYCLIC` distributions	+	+	+
`GEN_BLOCK`	+	–	+
`INDIRECT`	+	–	+
templates	–	+	+
`SHADOW` specifications	+	–	+
`HALO` specifications	+	–	–
distribution to processor sections	+	–	+
dynamic distributions	+	–	+
dynamic realignment	–	–	+
distributions of derived type components	+	–	+
distribution of pointers	+	–	+
Procedure interface			
inherited distributions	+	+	+
distribution ranges	+	–	+
prescriptive mappings	+	+	+
descriptive mappings	–	+	+
transcriptive mappings	–	+	+
procedures returning distributions	+	–	+
Other features			
work distribution for parallel loops	+	–	–
`PUREST` procedures	+	–	–
schedule `REUSE`	+	–	–
`EXTRINSIC` procedures	+	+	+

3.2. Data distribution features

In addition to the HPF-2 distribution mechanisms such as block, cyclic, block-cyclic distributions, and replication, VFC provides generalized block and indirect distributions. General block distributions provide enough flexibility to meet the demands of some irregular computations: if, for instance, the nodes of a simple unstructured mesh are partitioned prior to execution and then appropriately renumbered, the resulting distribution can be described in this manner. Indirect distributions offer the ability to express totally unstructured or irregular distributions that do not involve replication.

Distributions may be specified with respect to regular processor array sections. Simple alignments may be used to specify groups of arrays that are to be distributed together. Moreover, VFC implements the distribution of components of derived types.

Dummy arguments may either be explicitly distributed using the above mechanisms or may *inherit* their distribution from the corresponding actual arguments. All distributions mentioned above may be defined for statically allocated as well as *dynamically allocated* objects such as allocatable arrays or pointers.

Moreover, VFC supports *dynamic data redistribution* either by means of the `REDISTRIBUTE` directive or by redistribution as a result of procedure calls. While dynamic data distribution is a very flexible feature, its unrestricted use may have a severe impact on the performance of the generated parallel program. For example, certain important optimizations may not be performed if multiple distributions may *reach* a particular array reference. Since interprocedural reaching distribution analysis [22,29] and similar techniques are restricted, language support is needed, in order to provide the compiler with information about the set of different distributions a dynamically distributed array may be associated with at runtime. For this purpose the `RANGE` attribute can be used in order to supply the compiler with information on the set of different distributions a dynamically distributed array (or a dummy array with an inherited distribution) may be associated with. Fig. 1 shows some of the data distribution features supported by VFC.

3.3. Independent loops

The `INDEPENDENT` directive may be used to assert that the iterations of a subsequent `DO` loop are independent and, therefore may be executed in any order. In this context VFC supports the `NEW` clause for defining variables that are private to a loop iteration, the `REDUCTION` clause for performing reduction operations, and the `ON HOME` clause to specify how the loop iterations should be distributed.

3.4. Communication schedule reuse

Communication schedules are internal objects that have to be constructed by the runtime system when a parallel loop with irregular array accesses is entered. This is done in a pre-processing phase called inspector [30], which is usually very time-consuming. Informally, a communication schedule of a distributed ar-

```
!HPF$ PROCESSORS R(NP*NP)
!HPF$ PROCESSORS R2(NP,NP) = RESHAPE(R)

!     multidimensional distributions
!
!HPF$ DISTRIBUTE(BLOCK,*,CYCLIC)
      ONTO R2 :: A3

!     general block distribution
!
      INTEGER BS(4)
!HPF$ DISTRIBUTE(GEN_BLOCK(BS), *)
      ONTO R1 :: A2

!     indirect distributions
!
      INTEGER MAP(100)
!HPF$ DISTRIBUTE(INDIRECT(MAP),*) :: A2


!     distributed derived type components
!
!     TYPE GRID
         REAL, POINTER :: U(:,:), F(:,:)
!HPF$    DISTRIBUTE(BLOCK,BLOCK) :: U
!HPF$    ALIGN WITH U :: F
      END TYPE GRID

      TYPE (GRID), ALLOCATABLE :: MG(:)

!     dynamic distribution and
!     redistribution
!
!HPF$ DYNAMIC, DISTRIBUTE(BLOCK) :: A1
      ...

!HPF$ REDISTRIBUTE(CYCLIC(M)) :: A1
```

Fig. 1. HPF+ data distribution features supported by VFC. Besides the basic block and cyclic distributions VFC offers generalized block and indirect distributions, distributions to processor subsets, distribution of derived type components, and dynamic data distributions.

ray describes for each processor participating in the parallel execution of a loop the sets of array elements that have to be sent to or received from other processor in order to guarantee correct execution. During the inspector phase all indirect accesses to distributed arrays in a parallel loop are analyzed and, based on the access pattern, the data distribution, and the distribution of loop iterations, a communication schedule is derived.

In many applications, however, parallel loops are often enclosed in a sequential time-step loop and the associated communication schedules may be invariant over many or all time-steps. In such cases the preprocessing cost of the inspector can be amortized by computing the schedules once and reusing them across subsequent time-steps as long as the schedules are invariant.

```
  DO t = 1, max_time     ! time-step loop
     CALL Solve(F, X, cond)
     if (reconfigure()) then
        ...
        ! reset reuse condition
        cond = .false.
        else
        cond = .true.
     end if
  END DO

  SUBROUTINE Solve(F, X, cond)
!HPF$ INHERIT :: F, X
     ...
!HPF$ INDEPENDENT, ON HOME(F(X(I)),
      REUSE(cond)
      DO i = 1, n        ! irregular loop
         ...F(X(I))...
      END DO
  END SUBROUTINE Solve
```

Fig. 2. Conditional communication schedule reuse. This example shows how the communication schedules of an irregular loop may be reused depending on a logical condition. From a time-step loop a computational procedure `Solve` is called in which distributed arrays are accessed indirectly within an `INDEPENDENT` loop. The communication schedules required for this loop are computed when `Solve` is executed for the first time and are reused on subsequent incarnations of `Solve` as long as the logical variable `cond` is `true`. Whenever it is necessary to reconfigure the data domain by changing the indirection array `X`, the logical variable `cond` has to be set to `false`, resulting in a re-computation of the required communication schedules by executing the inspector phase again.

The recognition of invariant communication schedules is crucial in obtaining reasonable object code performance. However existing analysis techniques [28] are not powerful enough to detect all instances of redundant schedule computations; runtime analysis techniques [30] are similarly restricted. Thus, language support is required for this problem.

VFC currently supports the reuse of communication schedules based on the HPF+ REUSE clause. The REUSE clause may be specified in the context of an independent loop to indicate to the compiler that the access patterns as well as the distribution of all distributed arrays accessed in the loop are invariant. As a consequence, the communication schedules associated with these arrays have to be computed only once and may be reused in subsequent executions of the loop. The reuse clause may be combined with a scalar logical expression in order to support conditional communication schedule reuse as shown in Fig. 2.

3.5. Halos

The (super) set of non-local elements of a distributed array that may be accessed by a processor during the execution of a program unit may be specified by means of *halo regions*. A halo region is an irregular, non-contiguous area describing the non-local access pattern of a particular processor. Halo regions are specified by means of the HALO attribute which may be combined with distribution specifications. Halos may be used for statically as well as dynamically distributed arrays. The halo of a dimension of a distributed array is usually specified by means of a Fortran function which, when evaluated on a particular processor, returns a one-dimensional integer array containing the global indices of non-local array elements accessed on that processor.

The information provided by halo regions can be utilized by the compiler to allocate memory for all non-local accesses, to establish a corresponding address translation scheme, and to compute the required communication schedule. In particular, the use of halo regions may improve the parallelization of irregular independent loops significantly by enabling the compiler to optimize the inspector phase of irregular loops.

Halos have been implemented in VFC in such a way that it is always the responsibility of the user to update the halo of distributed arrays by means of the UPDATE_HALO directive. Thus, this feature may not only be used to avoid redundant schedule computations but also to avoid redundant communication. An example of using a halo for a dynamically distributed array is shown in Fig. 3.

```
      INTERFACE
          FUNCTION halo_function(i, map)
             INTEGER i, map(:)
             INTEGER,
             POINTER :: halo_function(:)
          END FUNCTION halo_function
      END INTERFACE

!HPF$ PROCESSORS R(NUMBER_OF_PROCESSORS())
!HPF$ DYNAMIC :: A
      ...
!HPF$ REDISTRIBUTE (INDIRECT(map)), &
!HPF$ HALO (halo_func(I, map)) ON R(I) :: A
      ...
!HPF$ UPDATE_HALO :: A
```

Fig. 3. The HALO Directive. The set of non-local indices of array A that may be accessed by a particular processor R(I) is obtained by evaluating the array-valued function halo_func() on each processor with the corresponding processor index I. The execution of the high-level communication directive UPDATE_HALO results in communication to update the halo-area on each processor.

3.6. Locality assertions

In order to call procedures which operate exclusively on data private to a processor from within independent loops without introducing any overheads, VFC supports the concept of *purest procedures*. In Fortran 95, procedures that are called in the context of data parallel constructs must have the PURE attribute, which enforces that they are free of side-effects. However, for a parallelizing compiler the PURE attribute is of limited use, since a pure procedure may, for example, read distributed global data objects. Calling a procedure that accesses distributed global data objects from an independent loop may result in a serialization of the loop because communication may be required to reference the data. HPF+ extends the concept of pure procedures by providing *purest procedures*. By specifying the PUREST attribute, the user asserts to the compiler that the called procedure is pure (free of side effects) and that its invocation does not result in communication on entry, during execution, and upon exit from the procedure. This information, which is usually difficult to determine automatically, supports the compiler in generating efficient code for independent loops despite the presence of procedure calls.

The RESIDENT clause may be specified together with an ON HOME clause in the context of independent loops to indicate that a loop may be parallelized without the need for generating communication.

3.7. Extrinsics

VFC supports the extrinsic interfaces of HPF-2, HPF_LOCAL and HPF_SERIAL, by implementing the associated argument transfer mechanisms and the most basic functions from the HPF_LOCAL library such as MY_PROCESSOR, GLOBAL_SIZE, GLOBAL_SHAPE and a few others. By means of the extrinsic HPF_LOCAL interface HPF+ may be used in a more flexible way by allowing the user to switch from the global HPF+ view to a local view. Extrinsic procedures not only provide an escape mechanism from HPF+, for example to integrate program units that are written in an other paradigm such as message-passing, but also a means to integrate existing sequential code or to express coarse-grain data parallelism. This is discussed in more detail in Section 5.2.

4. Parallelization techniques

The parallelization strategy of VFC is based on the *SPMD* programming model. VFC translates a source program into an SPMD message-passing target program, which is parameterized in such a way that it can be executed on an arbitrary number of processors[3]. At runtime each processor of the target machine executes the same target program operating on its own data.

4.1. Overview

The support of advanced concepts, in particular dynamic redistribution and inheritance of distributions by dummy arrays, requires a parallelization strategy which is able to deal with *multiple reaching distributions*. In order to handle these features efficiently, VFC provides, besides additional language support and analysis techniques, a sophisticated management scheme for distributed arrays, as well as advanced parallelization strategies and runtime support.

Management of distributed arrays

VFC analyzes the distribution specifications and generates calls to runtime routines that initialize the *layout descriptor* of each distributed array. Layout descriptors contain all information concerning the distribution of an array (or a class of arrays) together with global and local shape information. In addition, layout descriptors store auxiliary information about buffers required for storing non-local data and communication schedules for updating halo areas.

Distributed arrays as supported by VFC may be statically or dynamically allocated and statically or dynamically distributed. A distributed array is either a *distributee* or an *alignee*. The distribution of arrays is usually parameterized with respect to the number of processors the program will be executed on based on the intrinsic function `NUMBER_OF_PROCESSORS()`. To support all these features, VFC transforms the input program in such a way that each distributed array is allocated at runtime at the earliest point at which both the shape and distribution information are available.

A statically distributed array has a fixed distribution within its scope of declaration. Thus, in the generated SPMD program, at the beginning of the executable section of the program unit where such an array is declared, the distribution or alignment specifications are evaluated and the results are stored in the array's layout descriptor. The array is allocated dynamically on each processor according to its local shape stored in the memory component of the layout descriptor, either at the beginning of the executable section, if it was statically allocated in the HPF+ program, or, if it is dynamically allocated, at the time of allocation when an `ALLOCATE` statement is executed for it. This process is first carried out for all distributees and then for all alignees of the corresponding program unit. Note that for a statically distributed array the distribution component of the layout descriptor does not change within the scope of the array's declaration, but the memory component may change if, for example, the shape of the array is modified.

A dynamically distributed array may change the distribution within its scope of declaration. Thus, whenever such an array is redistributed, the distribution component of the descriptor is updated and the array is reallocated with the new distribution, provided that its shape is known.

Transformation of procedure calls

In order to implement the transfer mechanisms of distributed arguments at procedure boundaries VFC analyzes all procedure calls and determines whether the distribution of an actual argument and the corresponding dummy argument are different, or whether a distributed actual argument is associated with a non-distributed actual argument or vice versa. This analysis utilizes information from interface blocks with the goal to avoid implicit redistributions or copy-in/copy-out argument transfers whenever possible.

In cases where actual and formal distribution match or the formal argument inherits the distribution, calls are transformed in such a way that each distributed argument is passed together with its layout descriptor without the need of redistribution or copying.

If, for example, a distributed actual argument is associated with a non-distributed formal argument, or a dummy with a different distribution, an implicit redistribution to replicate the actual argument is generated prior to the call, and after the call, if it is required to restore the original distribution. Whenever such a redistribution is generated, or arguments have to be copied, the user is informed about this potential runtime overhead by corresponding *diagnostic messages*.

4.2. Array assignments

Unlike most HPF compilers, array statements are not converted into loops, but are transformed by VFC

[3] This is only true if the program has been written in such a way that it is not dependent on a specific number of processors.

to array statements that operate on the local memory on each processor. The main advantage of this strategy is that the parallelism of the original construct is preserved. Therefore, both inter- and intra-processor parallelism can be exploited by the Fortran 90 compiler of the parallel target machine. Currently VFC distinguishes three different classes of array assignments which are handled with varying overheads.

The first class of array assignments can be handled in an optimal way without the need for any transformations. This requires that each dimension of all arrays on the right-hand-side with a non-scalar subscript is distributed in exactly the same way as the associated dimension of the left-hand-side array, and that all distributed dimensions of all arrays involved are accessed identically and as a whole.

The second class comprises array assignment statements for which only work distribution and index conversion has to be performed, but which can be processed without introducing inter-processor communication. This strategy is applied if, for all arrays on the right-hand-side, each dimension with a non-scalar subscript is distributed and accessed in exactly the same way as the associated dimension of the left-hand-side array. Such statements are transformed by adapting the section-subscripts of distributed dimensions in such a way that each processor only accesses its local data. These local section-subscripts are determined at runtime by intersection of the local index set of an array dimension with the original global section subscripts and converting the resulting section into local indices.

All other array assignment statements are transformed by introducing appropriate calls to runtime library routines that determine the work distribution and generate the required communication of non-local data into temporary arrays.

4.3. Parallelization of irregular loops

In contrast to most existing HPF compilers, VFC provides powerful runtime parallelization strategies for non-perfectly nested irregular independent loops. Such loop nests may access arrays that are distributed in more than one dimension and may contain conditional statements, reduction statements, and procedure calls. Previous work [9] in parallelizing irregular loops was based on linearization of arrays with multidimensional distributions. Our approach for handling arrays with multi-dimensional distributions is characterized by a dimension-wise processing of array subscripts and hence does not incorporate such aggressive program restructuring as caused by linearization.

In general, irregular loop nests are transformed by VFC based on the Inspector/Executor [16] paradigm into three phases: the *work distribution phase*, the *inspector phase* and the final *executor phase*. If distributed arrays appear in the body of conditional-statements, or if they are accessed by means of distributed indirection arrays, VFC applies different variants of this strategy. In addition, the information provided by means of the `HALO` directive or the `REUSE` and `RESIDENT` clause is utilized by VFC to perform crucial optimizations.

Work distribution

In this phase the iteration space of a loop nest is distributed to the available processors based on the `ON HOME` clause. If no `ON HOME` clause has been specified it is determined automatically based on simple heuristics. Depending on the distribution of the array in the `ON HOME` clause, different work distribution strategies are applied. For regular distributions (i.e., `BLOCK(M)`, `CYCLIC(M)` and `GEN_BLOCK`) the distribution of the loop iterations is determined by intersecting the iteration space of each loop with the local index set of the array in the on-clause using analytical formulas [7]. For irregular distributions the local iterations on each processor are determined by evaluating the `ON HOME` clause for all iterations of the loop.

Inspector

The major task of the inspector is to determine the access patterns of distributed arrays in a loop nest and to construct the corresponding communication schedules for non-local data accesses. Schedules are data structures representing the communication patterns required for exchanging non-local data between the processors participating in the execution of a parallel loop. The inspector code generated by VFC usually consist of a series of loops in order to evaluate and store the subscripts of all distributed arrays accessed in the loop. In order to minimize storage requirements access patterns of distributed arrays are represented on a dimension-by-dimension basis. The access patterns are passed to runtime procedures which determine the non-local data accesses, transform the original global indices into local indices, and construct the associated communication schedules required for communicating non-local data into buffers.

Multi-level indirections are handled by a multi-phase Inspector/Executor approach. First the schedules

are computed for all of the distributed index arrays at the deepest level. They are then used for gathering non-local data for the indirection arrays at the next higher level. This process is repeated until a top-level data array is reached, which is processed as described above.

Executor

In the executor phase all non-local data read in the loop nest are gathered according to the computed schedules, and the restructured loop nest is executed. In case non-local data has been written, a final scatter communication is performed.

VFC generates the executor by restructuring the original loop nest in several steps as follows. The iteration space of each independent loop is changed according to the work distribution. Subscript expressions in distributed dimensions of top-level arrays are transformed such that the original global indices are substituted by local indices. Finally, depending on whether non-local data is read or written, calls to gather and scatter communication routines are inserted before and after the loop nest, respectively.

Schedule reuse

The goal of schedule reuse is to reduce the overheads of runtime preprocessing performed in the work distribution and inspector phases of an irregular independent loop by reusing the information obtained during these phases across subsequent executions of the loop. However, in many cases, the compiler does not have sufficient information to decide whether an inspector computation is redundant. Therefore, eliminating redundant inspectors of irregular loops is performed in VFC based on the `REUSE` clause. For an independent loop with a `REUSE` clause the work distributor and inspector of a loop are guarded by means of conditional statements to enforce that they are executed only if the reuse-condition is true or if the loop is executed for the first time.

As the performance results in Section 5 clearly show, the reuse of communication schedules is the key feature in achieving acceptable performance for codes processed with runtime parallelization techniques. Note, however, that the reuse of communication schedules usually increases the memory requirements of a program, since data structures allocated during the preprocessing phases for storing access patterns and communication schedules can no longer be deallocated after an independent loop.

Optimizations based on halos and `RESIDENT`

For distributed arrays for which the user has specified the `HALO` attribute the inspector phase can be drastically simplified. A halo specifies all non-local references to a particular distributed array for the duration of the program execution. At the time the distribution of an array is determined, the information provided by a halo is used to allocate storage for non-local data on each processor and to compute a communication schedule. This schedule is then used for all occurrences of this array in the program. As a consequence, references to such an array do not require a full inspector phase; only the conversion from global to local indices has to be performed. The same holds for arrays for which the `RESIDENT` clause has been specified.

5. Performance results

We present performance results on two distributed memory machines, a QSW CS-2 with 32 SPARC processors (50 MHz) and a NEC Cenju-3 with 128 VR4400 (75 MHz) processors, for two kernels developed in the HPF+ project. The HPF+ kernels have been parallelized with VFC 1.1 and the generated message-passing programs have been compiled with the SUN F90 1.2 compiler on the QSW-CS-2 and with the NEC F90 2.1 compiler on the NEC Cenju-3, respectively.

In Figs 6 and 7 the performance of these kernels, both in terms of absolute times and speed-ups with respect to the sequential code versions, is compared to the performance of the HPF+ program compiled with the Fortran 90 compiler and executed sequentially, and to hand-written MPI message-passing programs. In order to assess the importance of the reuse of communication schedules Figs 6 and 7 also include the timings for a variant of the HPF+ kernels that did not reuse communication schedules.

5.1. Crash simulation kernel

The first kernel, developed by Guy Lonsdale, represents the basic stress-strain calculation of a crash-simulation code [15] based on 4-node shell elements. The kernel used in our evaluation employed an unstructured mesh with 25760 nodes and 25600 elements. The total size of the kernel is approximately 600 lines of code. A simplified structure of the kernel is shown in Fig. 4. Array `X` represents the coordinates of the nodal points, `IX` captures the connectivity of the elements in the unstructured mesh, and `F` and `V` represent

```
      REAL x(3,NNP), ix(4,NEL), f(6,NNP),
      v(6,NNP)
!HPF$ PROCESSORS R(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE(*,BLOCK) :: X, IX, F, V
      ...
      DO i=1, MAX_TIMES
         ...
         CALL update(f, v, x)
         CALL kforce(f, x, ix)
      END DO
      ...
  SUBROUTINE kforce(f, x, ix, ...)
      ...
!HPF$ INHERIT, RANGE(*,BLOCK) :: f, x, ix
      INTERFACE
         SUBROUTINE mforce(fn, xn)
!HPF$       PUREST
            ...
         END SUBROUTINE mforce
      END INTERFACE
      ...
!HPF$ INDEPENDENT, NEW(n1,..,fn,xn), &
!HPF$ REDUCTION(f), ON HOME(ix(1,i)), REUSE
      DO i = 1, NEL
         n1 = ix(1,iel)
         ...
         DO j = 1, 3
           xn(j,1) = x(j,n1)
           ...
         END DO
         CALL mforce(fn, xn)
         DO j = 1, 6
            f(j, n1) = f(j, n1) = fn(j, 1)
            ...
         END DO
      END DO
  END SUBROUTINE kforce
```

Fig. 4. Fragment from crash simulation kernel.

the forces and velocities at nodal points. The kernel is based on an explicit time-marching scheme which is represented by an outer time-step loop performing 1000 iterations. From within this loop, the subroutine KFORCE to calculate the shell-element forces and UPDATE to update the velocities and displacements according to the computed forces are called. The INDEPENDENT loop in KFORCE represents the element-wise force calculation. Before the call to MFORCE all required data is gather into private temporary variables which are declared as NEW in the INDEPENDENT directive. The communication required for gathering distributed data is determined at runtime by means of an inspector. The major part of the computational cost of the algorithm is within the procedure MFORCE which exclusively operates on processor-private data only, and thus does not require communication. This is indicated to the compiler by providing the PUREST attribute in the interface of MFORCE. After the call to MFORCE a sum-scatter reduction operation to add back the elemental forces to the forces stored in the nodes is performed.

The irregular loop in KFORCE is processed with the runtime parallelization techniques as described in Section 4.3. VFC generates two inspectors to generate the communication schedules required for accessing X and F, respectively. Since the loop iteration space, the distribution of F, X, and IX, and the contents of IX are invariant, these schedules, once computed, can be reused for all time-steps. In the code this is expressed by means of the REUSE clause.

In Fig. 6 two variants of the HPF+ kernel, with and without schedule reuse, are compared with the sequential Fortran 90 version (i.e., the HPF+ program compiled with the Fortran 90 compiler), with a sequential Fortran 77 version, and with a hand-written Fortran 77 message-passing program. The figures clearly show that acceptable performance of the HPF+ code can be achieved only in the case where the schedules of the irregular loop in MFORCE are computed only once and reused in the remaining 999 iterations. The gap between the version of the HPF+ kernel with and without schedule resue is more than a factor 10. Similar and even larger slow downs have been observed using commercial HPF compilers. Although the HPF+ kernel (with schedule reuse) shows a quite good scaling behaviour the total slow-down w.r.t the hand-coded MPI/F77 kernel is still about 2.5 on the CS-2 and about 2.2 on the Cenju-3.

This overhead results to a large extent from the less efficient executor code generated by the VFC compiler for the main computational loop. In this loop the MPI/F77 code utilizes temporary scalar variables to reduce the number of accesses to the indirection array IX. The VFC compiler, however, substitutes these scalar variables prior to parallelizing the loops, and does not yet perform common sub-expression elimination in the code generation phase. The worse scaling of the HPF+ kernel on more than 32 processors is mainly due to the communication overhead generated by the VFC compiler.

5.2. Weather prediction kernel

The second kernel, developed by George Mozdzynski, was extracted from the Integrated Weather Forecasting System (IFS) [2] of the European Centre for Medium Range Weather Forecasts (ECMWF). It

```
!HPF$ PROCESSORS R1(NP)
!HPF$ PROCESSORS R2(NP1,NP2)=RESHAPE(R1)
!HPF$ DISTRIBUTE(GEN_BLOCK(B),INDIRECT(MF)) :: ZGL
!HPF$ DISTRIBUTE(INDIRECT(MAPGP ),*) :: ZGA,ZGB

      ALLOCATE(ZGL(NPROMAG,NGT))
      ALLOCATE(ZGA(NGPTOTG,NGT), ZGB(NGPTOTG,NGT))
      ...
      DO I=1, MAX_TIMES

!HPF$   INDEPENDENT,NEW(J),REUSE(LREUSE)
       DO JFLD=1,NGT        ! GTOL transposition
!HPF$     INDEPENDENT
          DO J=1,NGPTOTG
             ZGL(INDL(J),JFLD)=ZGA(J,JFLD)
          ENDDO
       ENDDO

       CALL SIMFFT(ZGL,...) ! HPF_LOCAL routine

!HPF$   INDEPENDENT,NEW(J),REUSE(LREUSE)
       DO JFLD=1,NGT        ! LTOG transposition
!HPF$    INDEPENDENT
         DO J=1,NGPTOTG
           ZGB(J,JFLD)=ZGL(INDL(J),JFLD)
         ENDDO
       ENDDO

       CALL SIMPHYS(ZGB,...) ! HPF_LOCAL routine
       ...
      END DO
```

Fig. 5. Fragment from weather prediction kernel.

presents key issues arising in the representation of the IFS grid-point data spaces and the transpositions performed therein. As shown in Fig. 5, the kernel is characterized by two computational phases (simulated FFTs and physics) where no communication is required, and the two grid-point space transpositions, which require communication. grid point space (or physical space) is a quasi-regular (so-called reduced) 3 dimensional grid in which the number of grid points per latitude is reduced as the poles are approached. Grid point space cannot be represented by simple `BLOCK` or `CYCLIC` distributions without the severe overhead of additional interprocessor communication during the algorithmic stages. Instead a combination of `GEN_BLOCK` and `INDIRECT` distributions is required. As in the IFS production code, the computational routines to perform the Fast Fourier Transforms (`SIMFFT`) and to simulate the physics calculations (`SIMPHYS`) have been integrated based on the efficient `HPF_LOCAL` extrinsic mechanism. The subroutines `SIMPHYS` and `SIMFFT` are sequential routines which operate exclusively on data local to a processor and thus are not modified by the compiler. For the transpositions in grid point space nested `INDEPENDENT` loops with indirect array accesses were used. The communication schedules of these irregular loops were computed only once in the first time-step and reused in subsequent time-steps based on the information provided by the `REUSE` clause.

The kernel has been coded using modules with allocatable arrays. This dynamic structure allows to use the same executable for different resolutions, and resembles exactly the structure used in the IFS production code. The kernel used in our evaluation employed a grid point space at resolution T213 with a total of 134028 grid points and performed a total of 1000 time-steps. It consists of 42 source files with a total of approximately 5000 lines of code.

As shown in Fig. 7 the HPF+ kernel performs almost as well as the hand-coded message-passing kernel exhibiting a speed-up of 120 on 128 processors.

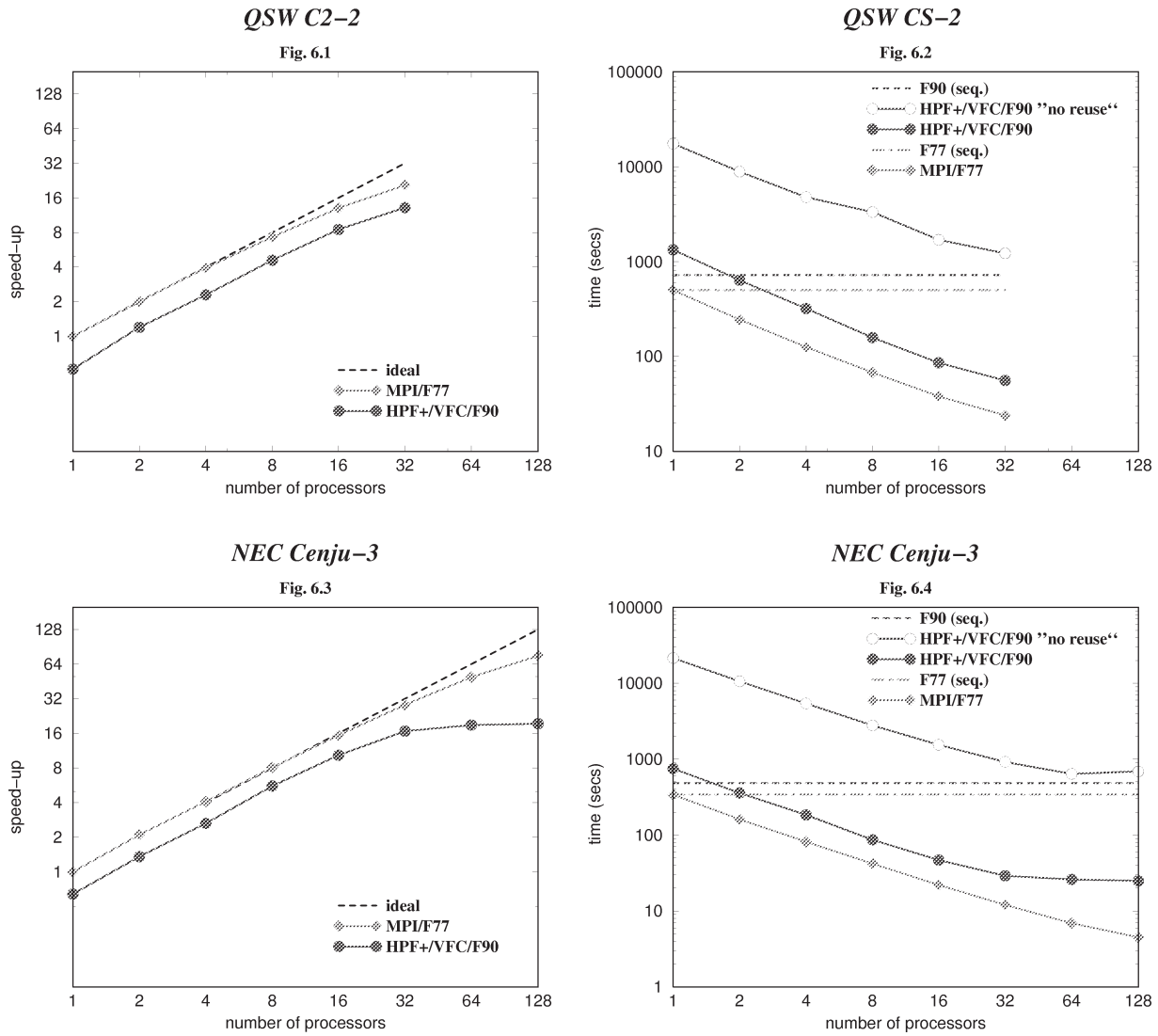

Fig. 6. HPF+ Crash Simulation Kernel. This diagram shows the speed-up and absolute run-times of the hand-coded MPI message-passing crash simulation kernel and the equivalent HPF+ kernel on the QSW CS-2 and the NEC Cenju-3. On the QSW CS-2 (Fig. 6.1 and 6.2) the MPI kernel was compiled with the SUN F77 compiler, while the VFC employed the SUN F90 to compile the generated message-passing program. Similarly, on the NEC Cenju-3 (Fig. 6.3 and 6.4) the MPI kernel was compiled with the F77 compiler while the VFC used the NEC F90 compiler to compile the parallelized HPF+ kernel. The speed-up of the HPF+ kernel has been computed with respect to the HPF+ program compiled with the F90 compiler of the target architecture and executed on a single node. The scalability obtained with HPF+ and VFC is quite satisfactory and close to the message-passing kernel. However, on the Cenju-3 it can be seen that scalability on more than 32 processors is significantly worse than that observed for the hand-written message-passing kernels. This is mainly due to the less efficient communication generated by VFC. In the hand-written message-passing code communication is performed on each processor only with its direct neighbors, whereas the code generated by VFC is based on all-to-all communication primitives. The elapsed times of the HPF+ kernel are approximately a factor 2.5 slower than those of the MPI kernel. Besides the all-to-all communication, this overhead results to a large extent from the less efficient executor code generated by VFC which does not employ common sub-expression elimination to reduce the number of accesses to the indirection array `IX`.

6. Related work

A number of research compilers for HPF and HPF-like languages have been or are being developed. In the following, some are briefly discussed.

The PARADIGM compiler [1] is a source-to-source parallelizing compiler based upon Parafrase-2 which accepts a sequential Fortran 77 or HPF program and produces an optimized message-passing Fortran 77 parallel program.

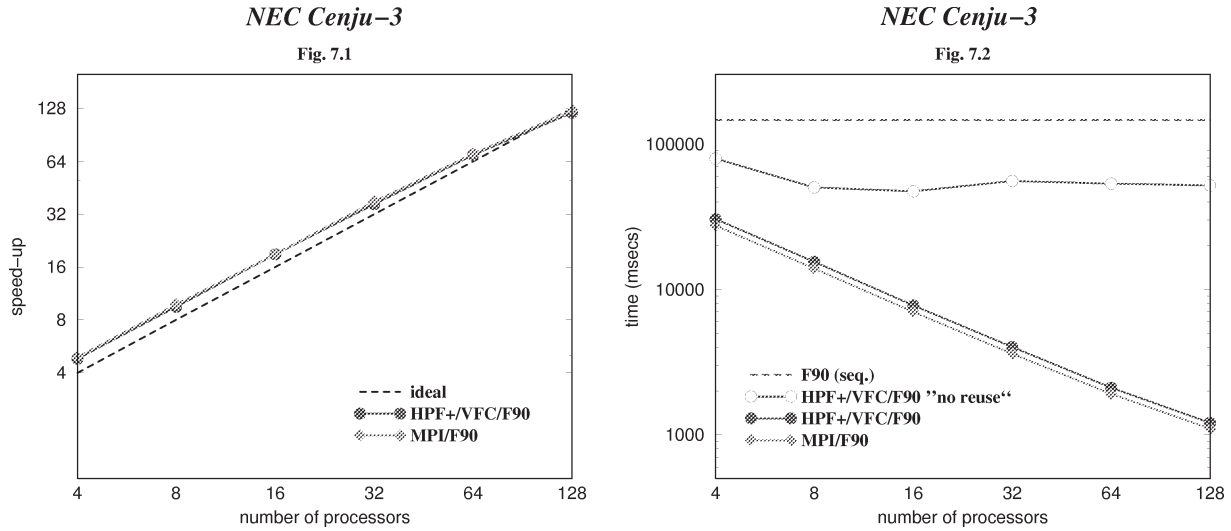

Fig. 7. HPF+ Weather Prediction Kernel. Figure 7.1 shows the speed-up of the hand-coded MPI message-passing and the equivalent HPF+ kernel with respect to the same serial version of the kernel. Fig. 7.2 shows the average time of 1 time-step of the serial kernel, of the HPF+ kernel without and with schedule resue, and of the hand-coded message-passing kernel. Unlike the crash-simulation kernels, the NEC Fortran 90 compiler was used to compile both the MPI message-passing program and the code generated by VFC. The HPF+ kernel performs almost as well as the message-passing kernel, which shows perfect scaling. The HPF+ code shows a speed-up of 120 on 128 processors. In terms of elapsed time the HPF+ kernel is only about 10% slower than the hand-coded MPI kernel. The HPF+ kernel could not be run on two processors due to memory constraints.

The sHPF system [25] transforms a Subset HPF program into a message-passing Fortran 90 program with calls to the ADLIB runtime library [12]. Parallel execution in sHPF is obtained from the use of Fortran 90 array syntax.

ADAPTOR [8] is a source-to-source transformation system for HPF that generates Fortran SPMD programs containing calls to the DALIB runtime system. Besides block and cyclic distributions ADAPTOR supports generalized block distributions. Moreover the system provides language support for reusing communication schedules based on the TRACE option, which allows the user to specify that the compiler should keep track of certain indirection arrays. Besides optimizing compilation techniques for regular loops and array assignments, ADAPTOR implements a limited form of the inspector/executor strategy. Unlike the VFC compiler which generates inspector code, ADAPTOR implements the inspector phase by calls to runtime library procedures. As a consequence, it can not handle non-perfectly nested loops or coupled array subscripts.

HPFC [23] is a Fortran 77 based prototype HPF compiler generating Fortran 77 message-passing programs with calls to a PVM based runtime library. HPFC provides extensions of HPF for the specification of local code regions. Although both ADAPTOR and HPFC provide preliminary support for dynamic redistribution, none of these systems can handle multiple reaching distributions.

The Fortran D compiler [22] converts Fortran D programs to F77 SPMD message-passing programs. Data distributions are restricted to single array dimensions and the number of processors has to be fixed at compile-time. Inspector/Executor parallelization techniques are applied for irregular computations.

The VFCS [5], the predecessor of VFC, is a source-to-source transformation system for Fortran 77 with Vienna Fortran and HPF extensions. VFCS supports besides block and generalized block distributions restricted forms of indirect distributions. Unlike VFC, VFCS supports only static distributions and requires the number of processor to be fixed at compile-time.

The PREPARE [9] compiler transforms a subset HPF-1 program into an SPMD message-passing program. Besides implementing a number of optimization techniques for regular codes, the system can handle limited forms of irregular loops based on the inspector/executor strategy.

Most of the above mentioned source-to-source compilers transform distributed multidimensional arrays into one-dimensional arrays. This linearization usually results in more complicated subscript expressions, and as a consequence, may inhibit important optimizations of the underlying target compiler. In contrast, VFC

transforms distributed arrays into ALLOCATABLE arrays. Accesses to distributed arrays are not linearized, but subscript expressions are translated in a dimension-wise fashion.

7. Conclusions

In this paper we presented the HPF+ compiler VFC, a source-to-source parallelization system that transforms Fortran95/HPF+ programs to Fortran 90 message-passing programs. VFC provides special support for an efficient handling of advanced codes based on highly irregular, possibly dynamically changing data structures and access patterns. VFC implements most features of HPF+, an optimized version of HPF for irregular codes. This includes block, cyclic, generalized block and indirect distributions, distributed components of derived types, processor views, distribution to processor subsets, dynamic data redistribution, and on-clauses. VFC offers powerful compilation strategies for nested loops with indirect accesses to arrays distributed in multiple dimensions, and performs important optimizations that decisively influence a program's performance such as the reuse of runtime generated communication schedules. Performance results with kernels extracted from advanced industrial applications indicate that the HPF paradigm, if suitably extended, may be successfully applied to irregular codes.

Future extensions of the VFC system will include additional analysis and optimization techniques. The parallelization strategy for irregular loop nests will be extended to loops with a non-rectilinear iteration space. Furthermore, distributed arrays of derived types will be implemented. This will include mechanisms for serializing derived type data objects. Task parallel features based on on-directives and task-regions [20] and coordination mechanism as proposed in OPUS [14] are currently being implemented. For an efficient handling of such features we plan to utilize *one-sided communication* primitives of MPI-2 [27].

Acknowledgment

The author would like to thank Viera Sipkova, Bob Velkov and Kamran Sanjari for their work in implementing the VFC compiler, and Guy Lonsdale and George Mozdzynski for providing the MPI and HPF+ kernels.

References

[1] P. Banerjee, J.A. Chandy, M. Gupta, E.W. Hodges, J.G. Holm, A. Lain, D.J. Palermo, S. Ramaswamy and E. Su, The PARADIGM compiler for distributed-memory multicomputers, *IEEE Computer* **28**(10) (1995).

[2] S. Barros, D. Dent, L. Isaksen, G. Robinson, G. Mozdzynski and F. Wollenweber, The IFS model: A parallel production weather code, *Parallel Computing* **21** (1995).

[3] S. Benkner, HPF+: High Performance Fortran for advanced industrial applications, in: *Proceedings International Conference on High Performance Computing and Networking (HPCN'98)*, Amsterdam, April 1998, Springer, Berlin.

[4] S. Benkner, P. Mehrotra, J. Van Rosendale and H. Zima, High-level management of communication schedules in HPF-like languages, in: *ACM Proceedings International Conference on Supercomputing*, Melbourne, Australia, July 1998. Also available as: NASA Contractor Report 201740, ICASE, Hampton, VA, September 1997.

[5] S. Benkner et al., *Vienna Fortran Compilation System. Version 1.2. User's Guide*, Univ. of Vienna, Inst. for Software Technology and Parallel Systems, February 1996.

[6] S. Benkner, Handling block-cyclic distributed arrays in Vienna Fortran 90, in: *IEEE Proceedings International Conference on Parallel Architectures and Compilation Techniques (PACT'95)*, Limassol, Cyprus, June 1995.

[7] S. Benkner, Vienna Fortran 90 and its compilation, Ph.D. Thesis. TR 94-8, University of Vienna, Institute for Software Technology and Parallel Systems, 1994.

[8] T. Brandes and F. Zimmermann, ADAPTOR – A transformation tool for HPF programs, in: *Programming Environments for Massively Parallel Distributed Systems*, Birkhäuser-Verlag, 1994, pp. 91–96.

[9] P. Brezany, O. Cheron, K. Sanjari and E. van Konijnenburg, Processing irregular codes containing arrays with multi-dimensional distributions by the PREPARE HPF compiler, in: *Proceedings International Conference on High Performance Computing and Networking*, Milan, May 1995.

[10] M. Calzarossa, L. Massari, A. Merlo, M. Pantano and D. Tessera, Integration of a compilation system and a performance tool: The HPF+ approach, in: *Proceedings International Conference on High Performance Computing and Networking*, Amsterdam, April 1998, Springer, Berlin.

[11] M. Calzarossa, L. Massari, A. Merlo, M. Pantano and D. Tessera, Medea: a tool for workload characterization of parallel systems, *IEEE Parallel Distrib. Technol.* **3**(4) (1995), 72–80.

[12] B. Carpenter, Adlib: A distributed array library to support HPF translation, in: *Proceedings 5th Workshop on Compilers for Parallel Computers*, Malaga, June 1995.

[13] B. Chapman, H. Zima and P. Mehrotra, Extending HPF for Advanced Data Parallel Applications, *IEEE Parallel Distrib. Technol.* (Fall 1994).

[14] B. Chapman, H. Zima, M. Haines, P. Mehrotra and J. Van Rosendale, OPUS: A coordination language for multidisciplinary applications, *J. Sci. Programming* (1995).

[15] J. Clinckemaillie, B. Elsner, G. Lonsdale, S. Meliciani, S. Vlachoutsis, F. de Bruyne and M. Holzner, Performance issues of the parallel PAM-CRASH code, *J. Supercomput. Appl. High-Performance Comput.* **11**(1) (1997).

[16] R. Das and J. Saltz, A manual for PARTI runtime primitives – Revision 2. Internal Research Report, University of Maryland, December 1992.

[17] R. Das, J. Saltz and R. von Hanxleden, Slicing analysis and indirect accesses to distributed arrays, in: *Proceedings 6th Workshop on Languages and Compilers for Parallel Computing*, August 1993, Springer, Berlin.

[18] J. Grosch and H. Engelmann, A tool box for compiler construction, in: *Lecture Notes Comput. Sci.*, Vol. 447, Springer, Berlin, 1990, pp. 106–116 .

[19] High Performance Fortran Forum, *High Performance Fortran Language Specification. Version 1.1*. TR, Rice University, November 10, 1994.

[20] High Performance Fortran Forum, *High Performance Fortran Language Specification. Version 2.0*. TR, Rice University, January 31, 1997.

[21] ISO, *Fortran 90 Standard*, May 1991, ISO/IEC 1539 :1991 (E).

[22] M.W. Hall, S. Hirandani, K. Kennedy and C.-W. Tseng, Interprocedural compilation of Fortran D for MIMD distributed-memory machines, in: *Proceedings of Supercomputing (SC92)*, Minneapolis, November 1992.

[23] R. Keryell, C. Ancourt, F. Coelho, B. Creusillet, F. Irigoin and P. Jouvelot, PIPS: A framework for building interprocedural compilers, parallelizers and optimizers, Technical Report 289, CRI, Ecole des Mines de Paris, April 1996.

[24] J. Knoop and E. Mehofer, Interprocedural distribution assignment placement: More than just enhancing intraprocedural placing techniques, in: *IEEE Proceedings International Conference on Parallel Architectures and Compilation Techniques (PACT'97)*, San Francisco, CA, November 1997.

[25] J.H. Merlin, D.B. Carpenter and A.J.G. Hey, SHPF: A Subset High Performance Fortran compilation system, *Fortran Journal* (March/April 1996), 2–6.

[26] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Vers. 1.1*, June 1995.

[27] Message Passing Interface Forum, *MPI-2: Extensions to the Message-Passing Interface*, 1997.

[28] R. von Hanxleden, K. Kennedy, C. Koelbel, R. Das and J. Saltz, Compiler analysis for irregular problems in Fortran D, in: *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*, New Haven, August 1992.

[29] D.J. Palermo, E.W. Hodges and P. Banerjee, Interprocedural array redistribution data-flow analysis, in: *Proceedings of the 9th Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, August 1996.

[30] R. Ponnusamy, J. Saltz and A. Choudhary, Runtime compilation techniques for data partitioning and communication schedule reuse, Technical Report, UMIACS-TR-93-32, University of Maryland, April 1993.

[31] H. Zima, P. Brezany, B. Chapman, P. Mehrotra and A. Schwald, Vienna Fortran – a language specification, ICASE Internal Report 21, ICASE, Hampton, VA, 1992.