High-Level Management of Communication Schedules in HPF-like Languages*

Siegfried Benkner[a] *Piyush Mehrotra*[b] *John Van Rosendale*[b] *Hans Zima*[a]

[a]Institute for Software Technology and Parallel Systems,
University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria
E-Mail: {sigi,zima}@par.univie.ac.at

[b]ICASE, MS 132C, NASA Langley Research Center, Hampton VA. 23681 USA
E-Mail: {pm,jvr}@icase.edu

Abstract

The goal of High Performance Fortran (HPF) is to "address the problems of writing data parallel programs where the distribution of data affects performance", providing the user with a high-level language interface for programming scalable parallel architectures and delegating to the compiler the task of producing an explicitly parallel message-passing program. For some applications, this approach may result in dramatic performance losses. An important example is the inspector/executor paradigm, which HPF uses to support irregular data accesses in parallel loops. In many cases, the compiler does not have sufficient information to decide whether an inspector computation is redundant or needs to be repeated. In such cases, the performance of the whole program may be significantly degraded.

In this paper, we describe an approach to solve this problem through the introduction of constructs allowing explicit manipulation of communication schedules at the HPF language level. The goal is to avoid the use of EXTRINSICS for expressing irregular computation via message-passing primitives, while guaranteeing essentially the same performance. These language features allow the user to control the reuse of schedules and to specify access patterns that may be used to compute a schedule. They are being implemented as part of the HPF+ language and we report some preliminary performance numbers from this implementation.

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 [11] focussed on extending Fortran 90 by directives for specifying alignment and regular data distributions (block and cyclic). 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. It soon became apparent that HPF-1 did not provide enough flexibility for an efficient formulation of many advanced algorithms. Such algorithms, for example weather forecasting codes or crash simulations, are often characterized by the need to distribute data in an irregular manner and dynamically balance the computational load of the processors. HPF-2 [12], together with its "approved extensions", is a significant step in supporting such algorithms.

*The work described in this paper was partially supported by the ESPRIT IV Long Term Research Project 21033 "HPF+" of the European Commission, by the Austrian Ministry for Science and Transport under contract GZ 613.580/2-IV/9/95, and by the National Aeronautics and Space Administration under NASA Contract No. NAS1-19480, while the last three authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23681.

HPF directives provide, at a high level of abstraction, information useful to the compiler in the context of a single-threaded data parallel paradigm with a global address space. It is 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. 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 "low-level" information that may decisively influence a program's performance cannot be expressed.

In this paper we focus on one such example – the efficiency of the communication required to handle irregular data accesses in parallel loops. Most compilers generate code which at runtime determines a communication schedule to handle the gathering and scattering of nonlocal data required for executing the parallel loop. The computation of such a schedule, based on a runtime analysis of the access patterns to an array, is performed by a routine called an *inspector*. Inspectors may be very time-consuming; as a consequence, the avoidance of redundant inspector executions is a high-priority goal.

In this paper, we support this goal by proposing a method which allows control of schedule computations at the HPF language level. Our method is based on the concept of *schedule variables*, whose values are communication schedules computed by an inspector or, alternatively, defined by the user by specifying the access pattern to an array with appropriate high-level directives.

Thus, using these features the programmer is not forced to switch to the message-passing paradigm via the **EXTRINSICS** facility of HPF, while obtaining basically the same performance as with message passing. The concepts described here are applicable to any HPF-like language; they are currently being implemented as part of the language *HPF+*, which was initially specified in [5] and is currently being developed into a fully-specified Fortran 95-based programming language in the ESPRIT Long Term Research Project "HPF+". The required compiler and runtime support is being implemented as a part of the *Vienna Fortran Compilation System (VFCS)* [2].

The paper is organized as follows. The next section provides a more detailed discussion of the inspector-executor paradigm and additional motivation for our work. Section 3 introduces the concepts and terminology required to describe schedules and access patterns. Section 4 introduces the new features to control the generation of schedules through a series of small examples while a more complete example is given in Section 5. Section 6 describes how users can specify the access patterns for a loop. Section 7 discusses the implementation of these features and provides some details of the performance we have gained when using these concepts in a compiler. The paper ends with an overview of related work (Section 8) and a short conclusion.

2 Motivation

In many cases – in particular, for regular numerical computations operating on dense data structures – an HPF compiler can statically determine the access patterns for arrays in the source program and, from this information, derive the required communication in the parallel target program. This is not possible for irregular problems, in which arrays are accessed via index arrays defined dynamically. For such problems, the access patterns as well as the associated communication schedules must be computed at runtime. The standard implementation technology for this situation is called the *inspector-executor paradigm* [13, 16, 17]. That is, for a parallel loop with indirect data accesses, an *inspector* analyzes the data access pattern at runtime. From the access pattern, the array distribution, and the work distribution of the loop, a communication schedule can be derived. This schedule is then used in the *executor* to actually perform the required communication for the loop by gathering all nonlocal data to be read in the loop at the beginning, and scattering all nonlocal data that were written in the loop, at the end.

The inspector phase for a parallel loop can be very expensive and may dominate the execution time for a whole program. However, the parallel loop is often enclosed in a sequential loop (for example, a time-step loop), and its communication schedule may be invariant over many or all of the iterations of the sequential loop. The recognition of invariant communication schedules is crucial for obtaining high object code performance. In some cases, a compiler can recognize this invariance automatically. However, often such loop structures are hidden in a hierarchy of procedure calls, so that even sophisticated interprocedural analysis may be unable to achieve this goal.

The main motivation for the language extensions described in this paper is to allow the user explicit control over the generation of communication schedules, thus avoiding the redundant computation of communication schedules by an inspector. We introduce *schedule functions* as mappings from a processor set to the powerset of an array index domain, describing the set of array elements that have to be communicated (read or written) to or from each processor. By combining schedule functions with a specific array and a *direction* (read or write), we obtain an *array schedule*, which is a precise specification of a gather or scatter communication for that array. We also introduce the concept of a *schedule variable* which can be used to name the schedule computed by an inspector. Using schedule variables, the user can explicitly control the reuse of schedules by directing the compiler to skip inspector analysis if a related schedule variable already contains the communication schedule required for executing the loop.

For parallel loops with complex bodies, possibly including nested loops, conditional statements and procedure calls, the inspector computation can become highly complex. If, in such a situation, the user is in a position to specify the access pattern of a loop, the inspector computation can be replaced by an evaluation of that pattern in the context of the loop. Our language extensions support this feature by introducing the concept of a *pattern function* – a mapping from a loop iteration range to the powerset of an array index domain. A pattern function can be combined with a specific loop and a specific array, to provide an *array loop pattern* which can be directly mapped to an array schedule. We provide a mechanism for the specification of *simple patterns* based on a generalization of Fortran 90 section notation, taking advantage of the fact that vector subscripts can be used to describe irregular access patterns. For more general cases, for example when access patterns depend on nested conditionals and loops, the absence of set types in Fortran enforces new syntax and semantics. We are currently studying mechanisms based on the concept of *user-defined pattern functions*, which allow the specification of arbitrary patterns and will report on such mechanisms in a future paper.

3 Schedules and Patterns: Concepts and Terminology

In this section, we introduce the concepts and terminology required for a discussion of communication schedules and access patterns. Assume for the following that A is an array with index domain $\mathbf{I}$, $\mathbf{P}$ denotes a set of processors, and L denotes a perfectly nested loop with iteration domain $\mathbf{R}$.

3.1 Data and Work Distributions

Definition 1 Data Distribution

1. *A (replication-free)* **(data) distribution** *for A with respect to $\mathbf{P}$ is a total function $\delta^A : \mathbf{I} \rightarrow \mathbf{P}$.*

2. *Let A, $\delta^A : \mathbf{I} \rightarrow \mathbf{P}$, and $p \in \mathbf{P}$ be given. Then $\lambda^A(p) := \{\mathbf{i} \in \mathbf{I} \mid \delta^A(\mathbf{i}) = p\}$ determines the set of elements of A owned by processor p under distribution δ^A. This is called the* **distribution segment** *of A with respect to p, its elements are the* **local variables** *of A in processor p.* □

Definition 2 Work Distribution

1. *A* **work distribution** *for L is a total function $\omega^L : \mathbf{R} \rightarrow \mathbf{P}$. For each $\mathbf{r} \in \mathbf{R}$, $\omega^L(\mathbf{r})$ specifies the processor in which iteration $\mathbf{r}$ is to be executed.*

2. *Let L, a work distribution ω^L, and $p \in \mathbf{P}$ be given. Then $\chi^L(p)$ denotes the set of all iterations of L to be executed in p: $\chi^L(p) := \{\mathbf{r} \in \mathbf{R} \mid \omega^L(r) = p\}$. $\chi^L(p)$ is called the* **execution set** *of p with respect to L.* □

3.2 Schedules

Definition 3 Schedule Functions and Array Schedules

1. *Let* $\mathbf{I}$ *denote an array index domain. A* **schedule function** *is a total function* $\sigma : \mathbf{P} \to \mathcal{P}(\mathbf{I})$.
2. *An* **array schedule** *for array A with distribution segment* λ^A *is a triplet* (A, σ, d), *where*
   (a) σ *is a schedule function with range* $\mathcal{P}(\mathbf{I})$
   (b) $d \in \{R, W\}$ *specifies a* **direction***:* read *("R"), or* write*)("W"). If d="R", then the array schedule is called a read or gather schedule, otherwise a write or scatter schedule.*
   (c) *for each* $p \in \mathbf{P} : \sigma(p) \cap \lambda^A(p) = \phi$. □

Schedule Application

Let $\bar{\sigma} = (A, \sigma, R)$ denote a read schedule. Then the *application* of $\bar{\sigma}$ results for each processor p in

- receiving all elements $A(\mathbf{i})$ with $\mathbf{i} \in \sigma(p)$. By definition, these elements are nonlocal with respect to p.
- sending all elements $A(\mathbf{i})$ local to p to all processors p' such that $\mathbf{i} \in \sigma(p')$.

Each processor p must establish a set of *buffer elements* in its local memory for storing the nonlocal objects specified in $\sigma(p)$.

Let p, p' denote two arbitrary distinct processors. Then we denote by *RECEIVE(p,p')* the set of elements of A to be received in p from p', and by *SEND(p,p'):=RECEIVE(p',p)* the set of elements of A to be sent from p to p'. Based on the above read schedule $\bar{\sigma}$, the receive sets can be immediately specified as *RECEIVE(p,p')*$:=\sigma(p) \cap \lambda^A(p')$. Due to the symmetry of the two classes of sets, we can then determine all send sets after a global communication phase.

The application of a write schedule is similar. Note that write schedules are only relevant if a work distribution different from the *owner computes paradigm* [20] is used.

3.3 Patterns

Schedules, as discussed above, are usually determined based on an inspector analysis of array access patterns in a loop. In this section, we formalize the concepts related to patterns and specify the mapping from patterns to schedules.

Definition 4 Pattern Functions and Array Loop Patterns

1. *Let* $\mathbf{R}$ *denote a loop iteration domain, and* $\mathbf{I}$ *an array index domain. A* **pattern function** *is a total function* $\pi : \mathbf{R} \to \mathcal{P}(\mathbf{I})$.
2. *An* **array loop pattern (ALP)** *for array A and loop L with direction d is a quadruplet* (L, A, π, d), *where* π *is a pattern function mapping the loop iteration domain of L to the index domain associated with A. An ALP is respectively called a read pattern or a write pattern, depending on whether* $d =$ *"R″ or* $d =$ *"W″.* □

A pattern function describes an array access pattern for the iterations of a parallel loop. More specifically, it expresses the fact that, for each $\mathbf{r} \in \mathbf{R}$, iteration $\mathbf{r}$ accesses the elements with indices $\mathbf{i} \in \pi(\mathbf{r})$ for some array. Note that π does not depend on a *particular* array or loop – it refers only to the associated index and iteration domains – and is also independent of related data and work distributions.

When we bind a pattern function to a specific array (with a well-defined distribution), a specific loop (with a well-defined work distribution), and a direction we obtain an ALP. From a given ALP, an array schedule can be determined according to the following lemma:

Lemma 1 Mapping ALPs to Array Schedules
Let (L, A, π, d) *denote an ALP. The corresponding array schedule is given by* (A, σ, d), *where* $\sigma = p2s(\omega^L, \delta^A, \pi)$ *is defined as*

$$\sigma(p) := \bigcup_{\mathbf{r}\in\chi(p)} \pi(\mathbf{r}) - \lambda(p) \text{ for each } p \in \mathbf{P}.$$

□

As can be seen from the construction used in the above lemma, the array schedule for a given ALP only depends on the associated data and work distributions. Two ALPs (L, A, π, d) and (L', A', π', d') are called **equivalent** iff

- the loop iteration domains and work distributions of L and L' are identical,
- the index domains and data distributions of A and A' are identical, and
- their pattern functions, π and π', are identical.

4 Language Features for Schedule Control

In this section, we describe the syntax and semantics of language extensions for the explicit control of schedules. The main objective for the introduction of these features is to provide the user with a means to assert to the compiler/runtime system that a schedule computation is redundant and can be suppressed, reusing a previously computed schedule.

4.1 Schedule Variables and Their Values

Schedule variables are declared using a **SCHEDULE** directive, which must occur in the specification part of a program unit.

!HPF+ **SCHEDULE** :: S

Schedule variables can be organized into arrays and appear as components of derived types. At any time within its scope, a schedule variable is either *undefined* or has a well-defined value. Initially – immediately after processing its declaration (or, if a **SAVE** attribute is specified, after processing the first instance of the declaration) – a schedule variable is set to *undefined*. Another way to set i a schedule variable to *undefined* is to apply the **RESET** directive to the variable.

At any time, a schedule variable is *associated* with a (possibly empty) set of array schedules, $\{(A_1, \sigma, d_1), \ldots, (A_n, \sigma, d_n)\}$, $n \geq 0$. If the schedule variable is *defined*, i.e. $n \geq 0$, then all A_i have the same index domain, $\mathbf{I}$, the same data distribution, δ, and the same communication behavior, as specified by σ. [1] We call the triplet $(\mathbf{I}, \delta, \sigma)$ the **core** of the value bound to the schedule variable.

A given array may occur in two array schedules associated with a schedule variable, if it uses a gather and a scatter schedule with the same schedule function.

If we discuss the **reuse** of a schedule via a schedule variable with a defined value, then only its core is relevant: the schedule variable may be associated with a new set of arrays and arbitrary directions, as long as its core remains the same.

In the following, we will often characterize the value of a schedule variable by a set of equivalent ALPs rather than by explicitly specifying it in the above sense. Since, by Lemma 1, these ALPs uniquely determine a core as well as a list of associated array schedules, this does not restrict the generality of the discussion.

Our language extensions provide multiple ways for specifying the value of a schedule variable. The most important method is to use the result of the (implicit) inspector analysis performed for a parallel loop. Another method is *schedule assignment*, which, in a way similar to conventional assignment, binds the result of a *schedule expression* to a schedule variable. Furthermore, there are methods for the explicit specification of patterns, discussed in Section 6.

[1] A given array may occur in two array schedules associated with a schedule variable, if it uses a gather and a scatter schedule with the same schedule function.

There is one particular context for a schedule variable, established by a parallel loop, which has a special semantics. We call this a **def-use context**. If a schedule variable occurring in such a context is *undefined*, then a value for the variable will be defined (for example, by executing an inspector or evaluating a pattern specification). If, on the other hand, the schedule variable is already defined, then the schedule computation will be suppressed and the schedule to which the variable is bound will be applied. The occurrence of a schedule variable in a def-use context is semantically correct only if the associated schedule function specifies precisely the communication that has to be performed.

For certain simple cases, the user can control the redundancy of a schedule computation without explicitly introducing schedule variables. As described in the next subsection, we provide a single keyword, **REUSE**, to support this functionality.

In the next few subsections, we use a series of examples to show how users can control the computation of the schedule in various situations. We always assume here that the original computation of a schedule is performed by an inspector.

4.2 Unnamed Schedules

In some situations, the implementation can be directed to reuse a set of schedules without the need to explicitly introduce schedule variables for that purpose. In particular, if a schedule is to be reused with the same loop being called repeatedly and is not used with any other loop the user does not need to explicitly name the schedule and can control the reuse of the schedule for the loop by a *reuse clause*. In this case, *all* communication schedules required for the loop are treated as a single unit and are subject to the reuse semantics.

Example 1 Unconditional Schedule Reuse Without Schedule Variable

```
      DO t=1,max_time
        ...
!HPF+ INDEPENDENT, ON HOME(Y(I)), REUSE
        L: DO I=1,N
             ...
           X(IX1(I)) = X(IX2(I))+Y(I)*Y(I)/(Z(IX1(I))-U(IX1(I),IX2(I)))
             ...
           END DO
        ...
      END DO
```

In the above code, loop L, is executed max_times*. By specifying the* **REUSE** *attribute in the* **INDEPENDENT** *directive, the user is indicating that the access patterns for all arrays are not going to change across multiple executions of the loops. That is, the values of the arrays used as index vectors,* IX1 *and* IX2*, along with the distributions of the arrays* X, Y *and* Z *remain invariant during the execution of the outer loop. Thus, loop L, when entered first, will activate inspector analysis to determine the read patterns for X, Y, Z, and U, and the write pattern for X. For all subsequent executions, the schedules determined during the first execution will be reused.* □

In our experience, many codes which use indirection vectors to access arrays do not change the values of the vectors during execution once they have been initialized. For example, an unstructured grid code which is non-adaptive will read in the grid points and set up the grid interconnections in the initialization phase of the program. Since the grid does not change during the execution of the program these interconnections and consequently the index vectors representing the neighbors remain invariant throughout the program. In such situations, the **REUSE** construct provides the user with a simple mechanism to assert this fact to the compiler, which then can organize the reuse of the communication schedules once computed for the loop. Note that the directive will work even if the outer loop is in one procedure which repeatedly calls another procedure containing the the parallel loop. That is, in the above code, the loop L can be in a different procedure which is called from the outer loop.

In cases when the index vectors are changing or the arrays themselves are being dynamically redistributed, a simple generalization of the reuse construct allows the specification of a condition for reuse. This condition is evaluated whenever a schedule for the loop has already been computed by a previous execution. If it yields **TRUE**, the old schedule is reused; otherwise a new schedule is computed.

Example 2 Conditional Schedule Reuse Without Schedule Variable

```
      LOGICAL USE_OLD
      DO t=1,max_time
         ...
  !HPF+ INDEPENDENT, ON HOME(C(I)), REUSE(USE_OLD)
       L: DO I=1,N
          ...
             A(I) = B(IX(I)) + C(I)
          ...
        END DO
         ...
      USE_OLD = .TRUE.
      IF RECONFIGURE(...)
         THEN CALL RECOMPUTE(IX); USE_OLD=.FALSE.
      END IF
      END DO
```

Here, the logical variable USE_OLD *is used to further control the generation of the schedule. That is, its value is set to FALSE when the index vector, IX is changed. Note that if the schedule is undefined, which will be the case the first time that the loop L is executed, the value of the associated condition is ignored. In subsequent executions, the value of the condition, the logical variable here, will control the generation of a new schedule.* □

4.3 Using Schedule Variables

In this section, we illustrate the use of schedule variables for the specification of schedule reuse. We cover conditional and unconditional reuse, and reuse across different loops.

Schedule variables may be associated with arrays in the context of a parallel loop by means of the *gather directive* and the *scatter directive*. Both directives establish a def-use context.

The gather directive, in its simplest form, has the syntax

$$\textbf{GATHER}\,(A_1, \ldots, A_n :: S)$$

where S is a schedule variable, and $A_1, \ldots, A_n$ denote arrays with identical shapes and distributions, and equivalent read patterns in the loop. This construct associates S with the corresponding set of equivalent ALPs. [2]

Syntax and semantics for the scatter directive are similar, except that the keyword **SCATTER** is used and the direction of the communication is reversed.

No array may appear in more than one gather or more than one scatter directive associated with the same loop. Note, however, that a given array may be associated with different schedule functions in a gather and scatter schedule.

Example 3 Unconditional Schedule Reuse in a Loop

```
  !HPF+ SCHEDULE :: S
```

[2] This association of S can be extended by other gather or scatter directives in the context of the same loop.

```
      ...
   DO t=1,max_time
      ...
!HPF+ INDEPENDENT, ON HOME(C(I)), GATHER (B::S)
   L: DO I=1,N
         ...
         A(I) = B(IX(I)) + C(I)
         ...
      END DO
      ...
   END DO
```

At the time execution reaches the independent loop first, the value of S is undefined. At that time, the **GATHER** *clause causes the execution of the inspector for the accesses to B in the loop, and the assignment of the resulting schedule to S. On subsequent iterations of the outer loop, the schedule bound to S is reused for B.*

The work distribution of L, as specified by the **ON** *clause, is given by $\omega^L(I) = \delta^C(I)$ for all $I = 1, \ldots N$, i.e., iteration I is performed on the processor that owns element I of array C. The ALP for B is given as (L, B, π, R), where the pattern function, π, is specified as $\pi(I) = \{IX(I)\}$ for all I. The resulting array schedule for B is (B, σ, R), where (see Lemma 1) for each $p \in \mathbf{P}$:*

$$\sigma(p) := \bigcup_{I \in \chi^L(p)} IX(I) - \lambda^B(p).$$

□

If a schedule is to be reused across subsequent procedure invocations, it has to be declared with the save attribute. This is shown in the following example, which uses a slightly more general access pattern: the pattern function, π, for B is now $\pi(I) = \{IX1(I), IX2(I)\}$.

Example 4 Reusing Schedules in Procedures

```
   DO t=1,max_time
      ...
      CALL IRREG(B,IX1,IX2)
      ...
   END DO

   SUBROUTINE IRREG(X,IX1,IX2)
!HPF+    SCHEDULE, SAVE :: S
      ...
!HPF+    INDEPENDENT, ON HOME(C(I)), GATHER (B::S)
   L: DO I=1,N
         ...
         A(I) = B(IX1(I)) + B(IX2(I))*C(I)
         ...
      END DO
      ...
   END SUBROUTINE
```

□

Another mechanism for reusing schedules in procedures is to declare them as global variables in modules, thus implicitly saving them between calls. If schedules became first class objects in the base language, then they could be declared at an appropriate level and passed down the call chain similar to any other data structure.

By means of the *reset* directive, the definition status of a schedule variable may be set to *undefined*. This allows the control of schedule reuse depending on runtime conditions. We slightly modify Example 3 to illustrate this.

Example 5 Conditional Schedule Reuse

```
!HPF+ SCHEDULE :: S
   ...
      DO t=1,max_time
         ...
!HPF+ INDEPENDENT, ON HOME(C(I)), GATHER (B::S)
   L: DO I=1,N
         ...
         A(I) = B(IX(I)) + C(I)
         ...
      END DO
         ...
      IF RECONFIGURE(...) THEN
         CALL RECOMPUTE(IX)
!HPF+ RESET S
      END IF
      END DO
```

As in Example 3, when the independent loop is encountered first during execution, the value of S is undefined, and the gather clause results in the execution of the inspector and the assignment of the resulting schedule to S. On subsequent iterations of the outer loop, the value bound to S is reused as long as the logical function RECONFIGURE *yields* **FALSE***. If the reference to* RECONFIGURE *evaluates to* **TRUE***, the reset directive sets the definition status of S to* undefined*, and the inspector is executed anew when the independent loop is encountered in the next iteration of the outer loop.* □

The previous examples have shown how a single array can be associated with a schedule variable. Below, we generalize Example 3 by showing how to associate the same schedule variable with different arrays. If, in such a situation, an inspector has to be executed, the implementation selects one of the arrays for performing the inspector analysis. The language does not specify the selection criterion.

Example 6 Multiple Use of a Schedule

```
!HPF+ SCHEDULE :: S

         ...
      DO t=1,max_time
         ...
!HPF+ INDEPENDENT, ON HOME(C(IX(I))), SCATTER(A::S), GATHER(B,C::S)
   L: DO I=1,N
       ...
         A(IX(I)) = B(IX(I)) + C(IX(I))
       ...
      END DO
         ...
    END DO
```

This example differs from Example 3 in that all three arrays are accessed irregularly, using the same indirection array IX *and thus the same pattern function, based on the assumption that all three arrays are distributed identically. When execution encounters the independent loop first, S is* undefined. *One of the three arrays is then selected for inspector analysis.* □

Example 7 Using Multiple Schedule Variables in a Loop

```
DO t=1,max_time
    ...
  CALL SUB(X,Y,Z,U,V,IX1,IX2)
    ...
END DO

SUBROUTINE SUB(X,Y,Z,U,V,IX1,IX2)
!HPF+ SCHEDULE, SAVE :: S1,S2,S3
  ...
!HPF+ INDEPENDENT, ON HOME(Y(I)), GATHER (X::S2,Z::S1,U::S3), SCATTER (X::S1)
  L: DO I=1,N
      ...
     X(IX1(I)) = X(IX2(I))+Y(I)*Y(I)/(Z(IX1(I))-U(IX1(I),IX2(I)))
      ...
     END DO
      ...
END SUBROUTINE SUB
```

In this example

- *S1 is associated with the ALPs (L, Z, π_1, R) and (L, X, π_1, W), where $\pi_1(I) = \{IX1(I)\}$ for $I = 1, \ldots N$.*
- *S2 is associated with the ALP $\{(L, X, \pi_2, R)\}$, where $\pi_2(I) = \{IX2(I)\}$ for $I = 1, \ldots N$.*
- *S3 is associated with the ALP $\{(L, U, \pi_3, R)\}$, where $\pi_3(I) = \{IX1(I), IX2(I)\}$ for $I = 1, \ldots N$.*

The loop distribution is determined by the loop iteration range $[1 : N]$ and the associated on clause **ON** *HOME(Y(I)) (see Example 3).*

This example is only correct if $\delta^X = \delta^Z = \delta^V$. □

In the following, we extend Example 7 by showing how to apply a schedule computed in one loop to another loop.

Example 8 Using a Schedule Across Different Loops

```
!HPF+ SCHEDULE :: S1,S2,S3
   ...
!HPF+ INDEPENDENT, ON
HOME(Y(I)), GATHER (X::S2,Z::S1,U::S3), SCATTER (X::S1)
  L1: DO I=1,N
       ...
      X(IX1(I)) = X(IX2(I))+Y(I)*Y(I)/(Z(IX1(I))-U(IX1(I),IX2(I)))
       ...
      END DO
       ...
!HPF+ INDEPENDENT, ON HOME(Y(I)), GATHER (X,V::S1), SCATTER (V::S2)
  L2: DO I=1,N
```

```
   ...
   V(IX2(I)) = X(IX1(I))+Y(I)*Y(I)*V(IX1(I))
   ...
  END DO
 ...
```

Here, the values to which $S1$, $S2$, and $S3$ are bound via the gather and scatter clauses associated with $L1$ are the same as for loop L in the previous example. $S1$ and $S2$ are reused in $L2$. When $L2$ is entered for the first time, all schedule variables used in this loop already have a defined value, and no inspector analysis is required. □.

By reusing a schedule variable in two different loops, the user is guaranteeing that the core of the schedule, i.e., the index domain, the schedule function and the data distributions involved are the same. The compiler (and the runtime system) just reuse the schedules as specified without checking. Thus, the reuse of $S1$ and $S2$ in the above example would not work if the on clauses specified in $L1$ and $L2$ were different or if the distributions of X and V were not the same.

4.4 Schedule Assignment Directive

A *schedule assignment directive* has the form S = *schedule-expression*, where S is a reference to a schedule variable. Such an assignment is executed by evaluating the *schedule-expression* and binding its result to S.

For the purpose of this paper, we consider only two cases for a schedule expression: first, a reference to a schedule variable, and second, a *union* of schedules, which uses the operator symbol "+".

If the schedule expression is a reference to a schedule variable, then the result of evaluating it is the value to which the variable is currently bound.

If the schedule expression is a union, $S1 + S2$, then assume first that $S1$ and $S2$ both are *defined*. $S1$ and $S2$ must be associated with the same array index domain, $\mathbf{I}$, and the same array distribution, δ, resulting in respective cores $(\mathbf{I}, \delta, \sigma_1)$ and $(\mathbf{I}, \delta, \sigma_2)$. Then the value yielded by the expression is characterized by the core $(\mathbf{I}, \delta, \sigma_1 \cup \sigma_2)$, and the set of associated array schedules is empty. If $S1$ or $S2$ are *undefined*, or the expression is not well-defined, then the schedule expression returns *undefined*.

Example 9 Schedule Assignment

```
!HPF+ SCHEDULE :: S1,S2,S3
    ...
!HPF+ INDEPENDENT, ON HOME(Y(I)), GATHER(X::S2,Z::S1), SCATTER(X::S1)
   L1: DO I=1,N
      ...
     X(IX1(I)) = X(IX2(I))+(Z(IX1(I))
      ...
     END DO
      ...
!HPF+   S3=S1+S2  ! Schedule assignment, computing the union of schedules S1 and S2
!HPF+ INDEPENDENT, ON HOME(Y(I)), GATHER(X::S1,V::S3), SCATTER(V::S2)
   L2: DO I=1,N
      ...
     V(IX2(I)) = X(IX1(I))+Y(I)*Y(I)*(V(IX1(I)+V(IX2(I)))
      ...
     END DO
```

The schedule variable S3 is defined by a schedule assignment, which computes the union of the schedules associated with $S1$ and $S2$. S3 is associated with the ALP $(L2, V, \pi_3, R)$, where $\pi_3 = \pi_1 \cup \pi_2$.

```
! Type Declarations

TYPE vert
    INTEGER id
    INTEGER bdry_flag            ! boundary flag
    INTEGER par_a,par_b,par_c    ! vertices of parent cell
    REAL ca,cb,cc                ! interpolation coefficients
    REAL sol                     ! current solution
    REAL old_sol                 ! last solution
    REAL delta                   ! change in solution
    REAL res                     ! residual
    REAL f                       ! forcing function
END TYPE vert

TYPE edge
    INTEGER id
    INTEGER va, vb               ! vertices
END TYPE edge

TYPE grid_type
   INTEGER nedge
   INTEGER nvert
   TYPE(edge), POINTER, DIMENSION(:) :: elist
   TYPE(vert), POINTER, DIMENSION(:) :: vlist
END TYPE grid_type
```

Figure 1: Multigrid on an Unstructured Mesh: Type Declarations

5 Unstructured Mesh Multigrid Example

In this section we present, in relative detail, a more concrete example, using multigrid techniques on an unstructured mesh. We show using the features described above would allow user to specify reuse of the communication schedules for the multiple levels of the unstructured mesh. Note that the code shown here is not complete for the sake of brevity and clarity. In particular, we do not show any of the distributions since the discussion here is independent of the actual distribution used for the data structures.

Figure 1 shows the global type declarations used for an unstructured mesh including those for a vertex containing indices of the parent cell in a finer mesh, an edge with its two vertices, and a grid containing a list of edges and and a list of vertices. The main program generates the unstructured meshes, $grids$, at $nlev+1$ levels setting up the interconnections between the meshes and initializing the forcing function of the mesh at the finest level, i.e., $grids(0)$. It repeatedly calls the routine *cycle* to execute one multigrid cycle. The routine *cycle* (also shown in Figure 2) conducts one V-cycle of the multigrid algorithm It "relaxes" the grid at each level while calling the *rest* to restrict the values from a fine grid to a coarse grid. Then, it uses the routine *prolo* to interpolate values from a coarse grid to a finer grid. The code until this point does not require any communication, except possible synchronization required to initially generate an unstructured mesh in parallel.

The routine *relax*, shown in Figure 3 performs a relaxation on a unstructured mesh by sweeping over the edges of the mesh and updating the values at the vertices of each edge based on the old values at the vertices. The loop is declared to be *independent* and the each iteration is to be executed on the processor which owns the edge being computed upon. This could require gathering up the values at vertices which do not reside on the same processor as the edge, computing the new values and then scattering the values to the owning processors. Since each call to *relax* works on a mesh at a different level, an HPF compiler would have to

```
PROGRAM main
    INTEGER nlev
    TYPE(grid_type), ALLOCATABLE :: grids(:)
    REAL err

! allocates grids(0:nlev-1) and the edge list, elist, and the vertex list, vlist,
! at each level based on input values and sets up parent-child and neighbor interconnections
   CALL setup(grids, nlev)

! set fine grid forcing function
   CALL fine_set(grids(0)%vlist)

   DO WHILE (err .LE. 1.0E-5)
      CALL cycle(grids, nlev)
      err = sqnorm(grids(0)%vlist)
   END DO
   ...
END

SUBROUTINE cycle(grids, nlev)
     TYPE(grid_type) :: grids(:)
     INTEGER nlev
     INTEGER lv, lvc

! relax each level, and perform restriction, moving to coarsest grid
     DO lv = 0, nlev-1
        lvc = lv +1
        CALL relax(grids(lv)%vlist, grids(lv)%nvert, grids(lv)%elist, grids(lv)%nedge, lv, nlev)
        CALl rest(grids(lv)%vlist, grids(lv)%nvert, grids(lvc)%vlist, grids(lvc)%nvert, lv, nlev)
     END DO

! relax each level, and perform prolongations, moving to finest grid
     DO lv = nlev-1, 0, -1
        lvc = lv +1
        CALL relax(grids(lvc)%vlist, grids(lvc)%nvert, grids(lvc)%elist, grids(lvc)%nedge, lv, nlev)
        CALL prolo(grids(lv)%vlist, grids(lv)%nvert, grids(lvc)%vlist, grids(lvc)%nvert, lv, nlev)
     END DO
END
```

Figure 2: Multigrid on an Unstructured Mesh: Main Program and Cycle Routine

```
SUBROUTINE relax(vlist, nvert, elist, nedge, lv, nlev)
    TYPE (edge) :: elist(nedge)
    TYPE (vert) :: vlist(nvert)
    INTEGER nvert, nedge
    INTEGER lv, nlev

    INTEGER  ia, ib
    REAL om, flx
!HPF+ SCHEDULE, SAVE :: s(0:nlev-1), g(0:nlev-1)

! initialize residual to forcing function
    vlist(:)%res  = vlist(:)%f

! loop  over edges on this level accumulating  residual unto vertices
!HPF$ INDEPENDENT, ON (HOME(elist(e)), NEW (ia,ib,flx), REDUCTION (vlist)
!HPF+ GATHER (elist::s(lv)), GATHER (vlist::g(lv)), SCATTER (vlist::g(lv))
    DO  e = 1, nedge
       ia = elist(e)%va
       ib = elist(e)%vb
       flx  =  flux(vlist(ia), vlist(ib))

       vlist(ia)%res =   vlist(ia)%res + flx
       vlist(ib)%res =   vlist(ib)%res - flx
    END  DO

! loop over vertices on this level "relaxing" solution
     vlist(:)%sol  = vlist(:)%sol  + om * vlist(:)%res

! update boundaries
      CALL  apply_bc(vlist, nvert)
END
```

Figure 3: Multigrid on an Unstructured Mesh: Relaxation Routine

```
SUBROUTINE rest(fine_vlist, fine_nvert, vlist, nvert, lv, nlev)
    TYPE (vert) :: vlist(nvert), fine_vlist(fine_nvert)
    INTEGER lv, nlev, nvert, fine_nvert

    TYPE (vert) :: va, vb, vc
!HPF+ SCHEDULE, SAVE :: s(0:nlev-1), g(0:nlev-1)

! zero coarse grid right forcing function
   vlist(:)%f = 0.0
! loop  over fine grid vertices
!HPF$ INDEPENDENT, ON (HOME(fine_vlist(v)), NEW (va,vb,vc), REDUCTION (vlist(:) %res)
!HPF+ GATHER (fine_vlist::s(lv)), GATHER (vlist::g(lv)), SCATTER (vlist::g(lv))
    DO v = 1, fine_nvert
       va = fine_vlist(v)%par_a
       vb = fine_vlist(v)%par_b
       vc = fine_vlist(v)%par_c
!  accumulate residual at a fine grid vertex unto its coarse grid parent vertices
       vlist(va)%res = vlist(va)%res + fine_vlist(v)%ca * fine_vlist(v)%res
       vlist(vb)%res = vlist(vb)%res + fine_vlist(v)%cb * fine_vlist(v)%res
       vlist(vc)%res = vlist(vc)%res + fine_vlist(v)%cc * fine_vlist(v)%res
     END DO
END

SUBROUTINE prolo(fine_vlist, fine_nvert, vlist, nvert, lv, nlev)
    TYPE (vert) :: vlist(nvert), fine_vlist(fine_nvert)
    INTEGER lv, nlev, nvert, fine_nvert

    TYPE (vert) :: va, vb, vc
!HPF+ SCHEDULE, SAVE :: s(0:nlev-1), g(0:nlev-1)

    ! loop  over fine grid vertices
!HPF$ INDEPENDENT, ON (HOME(fine_vlist(v)), NEW (va,vb,vc), REDUCTION (fine_vlist(:)%sol)
!HPF+ GATHER (vlist::s(lv)), GATHER (fine_vlist::g(lv)), SCATTER (fine_vlist::g(lv))
    DO v = 1, fine_nvert
       va = fine_vlist(v)%par_a
       vb = fine_vlist(v)%par_b
       vc = fine_vlist(v)%par_c
! linearly interpolate values in vertices va,vb,vc and update v
       fine_vlist(v)%sol  = fine_vlist(v)%sol  +                  &
                 fine_vlist(v)%ca * vlist(va)%delta  +         &
                 fine_vlist(v)%cb * vlist(vb)%delta  +         &
                 fine_vlist(v)%cc * vlist(vc)%delta  +         &
                   v.ca*va.delta + v.cb*vb.delta + v.cc*vc.delta
    END DO
END
```

Figure 4: Multigrid on an Unstructured Mesh: Routines for Restriction and Prolongation

regenerate the schedule each time unless it could perform interprocedural analysis to determine that there are $nlev + 1$ different meshes each requiring a different schedule. Such analysis is complex and is currently not available in any HPF compiler.

Using the features described in the last section, the user can indicate the required number of schedules and when to reuse them. Thus, two arrays of $nlev + 1$ schedules are declared in the specification part. On each call to *relax*, the current level *lv* is also passed in and is used to index the appropriate schedule for gathering values from the *elist* and *vlist* and scattering data to the *vlist*. At each mesh level, the first call to *relax* would encounter an undefined schedule and hence the inspector would be called to generate the schedule. Since the schedules are declared with *save* attribute, on subsequent cycles of the multigrid, the old schedule at each level can be reused without executing the inspector. Again, if schedules were first class objects in the language, they could be declared at the top, e.g., in the *grid_type* itself, and then passed down the call chain.

Figure 4 shows the prolongation routine, *prolo* and the restriction routine, *rest*. These are similar to *relax* except they "translate" values between the vertex lists of meshes at two levels. The code structure is similar to the routine *relax* and similar declarations of schedule arrays can be used to specify the reuse of the schedules. The only difference is that we need on *nlev* schedules here since each schedule handles the data transfer between two levels. Again, without the schedule specification the compiler would either have to regenerate the schedules on each call or carry out complex inter-procedural analysis to determine the right schedule to use on each call.

6 Explicit Pattern Specification

Until now, we assumed that a schedule is originally computed by an inspector, which preprocesses the loop at runtime by analyzing its array access patterns. Based on the access pattern for an array, the inspector can determine the associated array schedule by taking into account the work distribution of the loop and the data distribution for the array.

Although, in principle, this approach is always possible, inspector analysis may become very costly if nested loops and procedure calls occur in the parallel loop. In this section, we discuss language features for *explicit pattern specification* to support the user in a situation where inspector computation is impractical or highly inefficient and the user knows the access patterns used in the application.

The evaluation of a pattern specification results in a *pattern function* (see Definition 4). In the context of a parallel loop, a pattern function can be combined with an array and a direction to form an ALP, from which a schedule can be determined.

We are currently studying how to extend the *simple patterns*, which can be specified based upon Fortran 90 array reference syntax, to provide a more general specification capability in a way similar to Vienna Fortran's user-defined distributions [19].

6.1 Simple Patterns

A **simple pattern** can be specified in the context of a gather or scatter clause associated with a parallel loop L. We extend the syntax of these clauses, as introduced in the last section, by allowing a pattern specification in the place of the schedule variable, possibly combined with an assignment to the schedule variable. We illustrate this using the gather clause.

GATHER $(A_1, \ldots, A_n$ :: [S =] *pattern-spec*)

The *pattern-spec* has the form

PATTERN (*pattern-element,...pattern-element*)

where each *pattern-element* is a parenthesized list of *section-subscripts*, one for each dimension of the arrays A_i, all of which must have the same rank. At least one section subscript must contain a reference to a *do variable* of L. Vector subscripts occurring in patterns are not restricted to one-dimensional arrays, as in

Fortran 90.

Let **R** denote the iteration domain of L, and **I** the common index domain of the A_i. Then a *pattern-spec* is evaluated in the following way:

1. Evaluate the *pattern elements*, yielding pattern functions $\pi_1, \ldots, \pi_m$, all of which which map **R** to the powerset of **I**.
2. Define the pattern function, π, for the *pattern-spec* as the union $\pi := \bigcup_{j=1..m} \pi_j$.
3. Bind π to all A_i: this produces a set of equivalent ALPs: $\{(L, A_1, \pi, R), \ldots, (L, A_n, \pi, R)\}$.
4. If a schedule variable, S, has been specified, then bind S to the set of ALPs determined above.

The pattern function for a pattern element is evaluated by determining the values of all variables occurring in the section subscripts, which are not do variables of L and replacing them by their values.

Example 10 Simple Pattern I

```
SUBROUTINE SWEEP(X,Y,EDGE)
      ...
!HPF+  SCHEDULE, SAVE :: S
      ...
!HPF+  INDEPENDENT, ON HOME(EDGE(I,1)), REDUCTION(Y), &
!HPF+  GATHER(X,Y::S=PATTERN((EDGE(I,1:2)))), SCATTER(Y::S)
     L: DO I=1,NEDGE
          N1=EDGE(I,1); N2=EDGE(I,2); DELTAX = F(X(N1),X(N2))
          Y(N1) = Y(N1) - DELTAX
          Y(N2) = Y(N2) + DELTAX
        END DO
       ...
END SUBROUTINE SWEEP
```

The simple pattern specification in this example consists of one pattern element, which in turn is a list with one section subscript, EDGE((I,1:2)*. This determines a pattern function,* π*, with* $\pi(I) = \{EDGE(I,1), EDGE(I,2)\}$ *for all* $I = 1, \ldots, NEDGE$*. The associated set of ALPs is* $\{(L, X, \pi, R), (L, Y, \pi, R), (L, Y, \pi, W)\}$*.*

This computation is only performed upon the first execution of L. All subsequent executions reuse this schedule. No inspector analysis has to be performed.□

The example below illustrates a pattern consisting of more than one pattern element. It expresses the patterns of the second loop in Example 8 explicitly, without using schedule variables.

Example 11 Simple Pattern II

```
    DO t=1,max_time
         ...
       CALL SUB2(X,Y,Z,U,V,IX1,IX2)
         ...
    END DO

SUBROUTINE SUB2(X,Y,Z,U,V,IX1,IX2)
!HPF+ INDEPENDENT, ON HOME(Y(I)), GATHER(X::PATTERN((IX1(I))), &
!HPF+ V::PATTERN((IX1(I), (IX2(I)))), SCATTER(V::PATTERN((IX2(I))))
   L2: DO I=1,N
```

```
    ...
    V(IX2(I)) = X(IX1(I))+Y(I)*Y(I)*(V(IX1(I)+V(IX2(I)))
    ...
    END DO
  ...
END SUBROUTINE SUB2
```
□

7 Implementation

In this section we describe the language features and compilation techniques provided by the Vienna Fortran Compilation System (VFCS [2]) for the parallelization of irregular applications. We outline the main steps in transforming independent do loops with irregular data accesses according to the inspector-executor strategy [17, 13, 16] and present performance results for a benchmark kernel which has been extracted from a crash-simulation code. A hand-optimized version of the parallel code shows that communication schedule reuse is crucial to amortize the overhead of this run-time parallelization strategy.

7.1 Inspector-Executor Parallelization Strategy

VFCS supports the HPF-2 `INDEPENDENT` directive together with the `NEW` and `REDUCTION` clauses. Moreover, the user may specify the mapping of loop iterations to processors by means of the `ON` and `ON HOME` clauses. Arrays accessed within independent loops may be distributed using the `BLOCK`, `GEN_BLOCK` or `INDIRECT` distribution formats. Indirection arrays may be distributed and arbitrarily nested.

The process of parallelizing irregular loops combines compile time parallelization techniques with runtime analysis. Information from data-flow analysis, data dependence analysis, data distribution and overlap analysis is utilized. The runtime support of VFCS for irregular loops is based on an enhanced version of the PARTI ([6]) and CHAOS ([18]) runtime routines.

VFCS transforms irregular independent loops into three main phases: the *work distributor*, the *inspector*, and the *executor*.

According to the on-clause of an independent loop L the work distributor calculates on each processor p the set $\chi^L(p)$ of loop iterations to be executed. Depending on its structure, the execution set is either represented as a regular section by a triplet or as a one-dimensional integer array.

The inspector performs a runtime analysis of the loop in order to determine for each distributed array the required communication schedules. This is accomplished by iterating on each processor over the computed execution set, evaluating the subscripts of all distributed arrays, and storing the indices in *global reference lists*. These lists are passed together with the corresponding *layout descriptors* to library procedures that return the resulting *gather/scatter schedules* and the corresponding *local reference lists*. In the case of a multi-level indirection, the array accesses are processed in multiple phases, starting with the innermost arrays.

For each nonlocal data item accessed in a processor, a local buffer element is reserved. The executor phase begins by gathering all nonlocal read data from remote processors, according to the gather schedules determined by the inspector, and placing them into the associated buffer elements. Following this transfer, the transformed loop body is executed; nonlocal data are read and written using their associated buffer elements. After all iterations have been performed, all nonlocal data that were written in a processor are scattered to their owner using the scatter schedules determined by the inspector.

7.2 Performance Results

We present performance results for a finite-element kernel that has been developed in the context of the HPF+ project. The kernel represents the basic stress-strain calculation of a crash-simulation code based on 4-node shell elements. It uses an explicit time-marching scheme which is represented in the kernel by an outer loop performing 250 iterations. A simplified structure of the kernel is shown in Figure 5.

¿From the time-step loop in the main program the subroutine `KFORCE` is called with distributed arguments. The corresponding actual arguments inherit the mapping and thus no data motion is required at the

Processors	Total (Unoptimized)	Inspector	Gather	Scatter	Speed Up
1 (seq.)	545.45	-	-	-	-
2	598.98	347.63 (58%)	2.31	2.38	0.9
4	304.75	172.48 (57%)	4.33	6.22	1.8
8	160.19	89.60 (56%)	5.33	7.81	3.4
16	100.04	50.66 (51%)	8.68	10.78	5.5
32	80.99	36.34 (45%)	10.78	11.65	6.7

Table 1: Times for crash kernel without schedule reuse.

procedure boundary. The main variables used in the kernel (`F, A`, and `V`) represent the forces, accelerations, and velocities at nodal points, respectively. `X` represents the coordinates of nodal points, and `IX` captures the connectivity of the elements in the unstructured mesh.

The first independent loop in `KFORCE` represents the element-wise force calculation. Before the call to `MFORCE` all required data is gathered into private temporary variables which are declared as `NEW` in the `INDEPENDENT` directive. The communication required for gathering distributed data into the temporaries is determined at runtime by means of inspectors. The major part of the computational cost of the algorithm is within the procedure `MFORCE` which operates on processor-private data only, and thus does not induce communication. The second independent loop performs a sum-scatter operation to add back the elemental forces to the forces stored at the nodes.

The kernel used in our evaluation employed an unstructured mesh with 35571 nodes and 35000 elements. A total of 20 distributed arrays were used for which VFCS generated 9 different inspector phases within the `KFORCE` procedure. Table 1 shows the time measurements obtained for the unoptimized kernel as parallelized by VFCS and executed on the Meiko CS-2 for 2 to 32 processors[3]. The second column of the table shows the total time spent in executing the procedure `KFORCE`. As can be seen, approximately 50 % of the total execution time are due to the overhead of the inspector phases for computing the required communication schedules. These schedules are recomputed in every incarnation of the procedure since the system fails to detect the invariance of the communication patterns. Note that in a real simulation the overhead induced by the repeated execution of inspectors would be orders of magnitude higher since the number of time-steps is usually in the range of 10^5.

Table 1 also shows the accumulated time spent with gather and scatter communications, respectively. Since the communication is unstructured and involves all processors, the fraction of time spent with communication increases almost linearly with the number of processors from less than 1% on 2 processors to 28% on 32 processors.

Table 2 shows the total time spent in procedure `KFORCE` where all communication schedules were computed in the first iteration of the time-step loop and reused in subsequent iterations. This optimization has been carried out by manually adapting the code generated by VFCS. As the timings show, the overhead of the time spent in computing communication schedules is reduced to less than 1% and results in a performance improvement of more than a factor of 2.

8 Related Work

The inspector/executor runtime preprocessing technique was initially implemented in the Kali [13] compiler. The PARTI [6] runtime library was developed to provide runtime support for a class of irregular problems characterized by a sequence of concurrent computational phases, where patterns of data access and computational cost of each phase cannot be predicted until runtime. They were designed to ease the implementation of irregular problems on distributed memory parallel architectures by relieving the user of having to deal

[3] The time for one processor refers to the HPF+ program compiled with the SUN FORTRAN 77 compiler Version 3.0.1 with optimization level -O3 and executed on a single node.

```
        PROGRAM
        ...
        REAL, DIMENSION(3,NUMNP) :: X
        REAL, DIMENSION(4,NUMEL) :: IX
        REAL, DIMENSION(6,NUMNP) :: F, A, V
        REAL, DIMENSION(6,NUMEL) :: FORCE1, FORCE2, FORCE3, FORCE4
!HPF$   DISTRIBUTE (*,BLOCK) :: X, IX, F, A, V, FORCE1, ...

        ...
        DO T=1,MAX_TIME
          ...
          CALL KFORCE(F,X,IX,...)
          ...
        END DO
        ...
        END


        SUBROUTINE KFORCE(f,x,ix,...)
!HPF$   INHERIT :: F, X, IX, ...
        ...
        REAL XN(3,4), FORCE(6,4)
        ...
!HPF+   INDEPENDENT, NEW(XN,FORCE,...), ON HOME(IX(1,I))
        DO I = 1, NUMEL
          XN = X(:,IX(:,I))                 ! gather coordinates

          CALL MFORCE(XN, FORCE, ...)       ! calculate forces

          FORCE1(:,i) = FORCE(:,1)
          ...
        END DO

!HPF$   INDEPENDENT, REDUCTION(F)
        DO I = 1, NUMEL                     ! sum-scatter reduction
           DO J = 1, 6
              F(J,IX(1,I)) = F(J,IX(1,I)) + FORCE1(J,I)
              ...
           END DO
        END DO
         ...
        END SUBROUTINE
```

Figure 5: Simplified structure of the finite-element crash simulation kernel.

Processors	Total (Schedule Reuse)	Inspector	Speed Up	Unoptimized/Schedule Reuse
2	285.22	1.47	1.9	2.1
4	153.00	0.81	3.6	2.0
8	79.10	0.47	6.9	2.0
16	47.47	0.31	11.5	2.1
32	35.31	0.20	15.4	2.3

Table 2: Times for crash kernel with schedule reuse (hand optimized).

with many low–level machine specific issues. The PARTI routines support communication schedule computation, global–to–local index transformation and schedule based communication generation. The CHAOS library [16, 18], a superset of PARTI, provides additional support for the parallelization of adaptive irregular problems where indirection arrays are modified during the course of the computation. A number of research compilers, including the Fortran D compiler [9], the Vienna Fortran Compilation System [2], and others [3] have used these libraries for the compilation of irregular codes based on the inspector/executor approach.

In [16] a simple run-time technique for communication schedule reuse is presented which is based on a global time-stamping method to keep track on whether indirection arrays that influence the communication pattern of a particular loop have been modified. Each inspector checks the corresponding time-stamps to determine whether relevant indirection arrays have been modified since the last inspector invocation.

Hanxleden [10] developed the Give-N-Take data flow framework which is used for the placement of communication statements in parallelized programs. He uses techniques that are based on partial redundancy elimination and symbolic analysis to eliminate redundant inspectors in certain restricted cases.

First proposals for applying program slicing techniques to the optimization of indirect array accesses were made in [7, 8]. These techniques construct program slices containing the subset of statements affecting nonlocal array accesses at a particular program point. Multiple indirection levels can be eliminated by applying a flattening transformation. A dedicated program analysis based on the topological sort of a slice graph is employed for the elimination of redundant slices. Most of these techniques, however, are restricted to certain limited forms of loop bodies and fail in the presence of procedure calls.

The PILAR [14] run-time support library developed in the context of the PARADIGM compiler [15] aims at exploiting regularity in irregular accesses by using an interval-based representation of communication schedules. The PARADIGM system relies on special directives to guide the compiler in placing the communication in the presence of irregular references.

9 Conclusion

In this paper, we described a set of language features that allow the explicit manipulation of communication schedules at the HPF language level. Our method is based on the concept of schedule variables, whose values are communication schedules computed by an inspector or, alternatively, defined by the user by specifying the access pattern to an array with appropriate high-level directives.

These features are currently being implemented in the framework of the Vienna Fortran Compilation System (VFCS). We plan to apply this methodology to a set of important applications, evaluate the resulting performance, and use the results to adjust the functionality of the language extensions accordingly.

Acknowledgment

The authors thank their partners in the ESPRIT project "HPF+", in particular Guy Lonsdale and George Mozdzynski, for many fruitful discussions on this subject. We also thank Viera Sipkova for the performance measurements discussed in Section 7.2.

References

[1] Benkner, S. Vienna Fortran 90 and its Compilation. Ph.D. Thesis. Technical Report TR 94-8, University of Vienna, Institute of Software Technology and Parallel Systems, November 1994.

[2] Benkner, S., Andel, S., Blasko, R., Brezany, P., Celic, A., Chapman, B.M., Egg, M., Fahringer, T., Hulman, J., Hou, Y., Kelc, E., Mehofer, E., Moritsch, H., Paul, M., Sanjari, K., Sipkova, V., Velkov, B., Wender, B., Zima, H.P.: Vienna Fortran Compilation System - Version 1.2 - User's Guide, October 1995.

[3] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF compiler for distributed memeory MIMD computers: Design, implementation, and performance results. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, July 1993.

[4] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming* 1(1):31-50, Fall 1992.

[5] B. Chapman, P. Mehrotra, and H. Zima. Extending HPF for Advanced Data Parallel Applications. IEEE Parallel and Distributed Technology, Fall 1994, pp.59-70.

[6] R. Das, J. Saltz, A manual for PARTI runtime primitives - Revision 2. Internal Research Report, University of Maryland, Dec. 1992

[7] R. Das, J. Saltz, and R. von Hanxleden. *Slicing Analysis and Indirect Accesses to Distributed Arrays.* Technical Report CS-TR-3076, UMIACS-TR-93-42, University of Maryland, College Park, MD, 1993.

[8] R. Das, A. Susman, P. Havlak, J. Saltz. *Compiler Analysis and Optimization of Indirect Array Accesses.* In *Proceedings of the 5th Workshop on Compilers for Parallel Computers.* Malaga, Spain, June 1995.

[9] 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, CT, August 1992.

[10] R. von Hanxleden and K. Kennedy. Give-N-Take: A balanced code placement framework. In *ACM SIGPLAN '94 Program Language Design and Implementation*, June 1994.

[11] High Performance Fortran Forum. High Performance Fortran Language Specification Version 1.0. Technical Report, Rice University, Houston, TX, May 3, 1993.

[12] High Performance Fortran Forum. High Performance Fortran Language Specification Version 2.0.Ω Technical Report, Rice University, Houston, TX, January 31, 1997.

[13] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[14] A. Lain and P. Banerjee. Exploiting Spatial Regularity in Irregular Iterative Applications. In *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, CA, April 1995.

[15] P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers, In *Proceedings of the First International Workshop on Parallel Processing*, Bangalore, India, December, 1994.

[16] R. Ponnusamy, J. Saltz, A. Choudhary. Runtime Compilation Techniques for Data Partitioning and Communication Schedule Reuse. Technical Report, UMIACS-TR-93-32, University of Maryland, April 1993.

[17] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(2):303–312, 1990.

[18] J. Saltz, R. Das, B. Moon, S. Sharma, Y-S. Hwang, R. Ponnusamy, M. Uysal : *A Manual for the CHAOS Runtime Library*, Technical Report, University of Maryland, College Park, MD 20742, May 1994.

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

[20] H. Zima and B. Chapman. Compiling for Distributed Memory Systems. *Proceedings of the IEEE*, Special Section on Languages and Compilers for Parallel Machines, pp. 264-287, February 1993.