Enlarging the Scope of Vector-Based Computations: Extending Fortran 90 by Nested Data Parallelism

K. T. P. Au*, M. M. T. Chakravarty‡, J. Darlington*, Y. Guo*
S. Jähnichen‡, M. Köhler*, G. Keller‡, W. Pfannenstiel‡, M. Simons‡

*Imperial College
Department of Computing
180 Queen's Gate
London SW7 2BZ, UK

‡Technische Universität Berlin
Forschungsgruppe Softwaretechnik
Franklinstr. 28/29 (FR5-6)
10587 Berlin, Germany

‡GMD FIRST
Rudower Chaussee 5
12489 Berlin, Germany

Abstract

This paper describes the integration of nested data parallelism into Fortran 90. Unlike flat data parallelism, nested data parallelism directly provides means for handling irregular data structures and certain forms of control parallelism, such as divide-and-conquer algorithms, thus enabling the programmer to express such algorithms far more naturally. Existing work deals with nested data parallelism in a functional environment, which does help avoid a set of problems, but makes efficient implementations more complicated. Moreover, functional languages are not readily accepted by programmers used to languages, such as Fortran and C, which are currently predominant in programming parallel machines. In this paper, we introduce the imperative data-parallel language Fortran 90V and give an overview of its implementation.

1 Introduction

Vector computers are one of the most successful architectures for high-performance computing. They offer fine-grain data parallelism, enabling one operation to be applied to many elements of a vector simultaneously. However, to be efficient, vector architectures require code that operates on regularly organised data, such as dense matrices or regular meshes. Consequently, the programming languages used for vector machines, such as Fortran, support parallelism only on rectangular array data structures in order to facilitate the generation of efficient target vector code. While this restriction in high-level programming languages stems mainly from limitations in compiler technology, it constitutes one of the major drawbacks of vector machines as against scalar parallel computers.

This problem can be resolved to some extent by developing programming and compiler technology that supports more flexible and structured data models. Recently, Blelloch [6] popularized the concept of *nested data parallelism*, which allows elements of a vector to be vectors (of arbitrary length) themselves. Hierarchical data structures can thus be uniformly coded in terms of nested vectors. This concept goes hand in hand with Blelloch's earlier technique of *flattening* nested data parallelism [4], which describes the transformation of nested vector computations into flat ones—thus enabling the power of vector architectures to be exploited for nested vector processing. The efficiency of the generated code has been improved in subsequent work [15, 17, 20]. Nested data-parallel languages such as Paralation-Lisp [22], NESL [9], and Proteus [13] have been proposed and implemented for a wide range of vector machines such as the CRAY Y-MP; SPMD machines such as the CM-5; and MIMD machines such as the IBM SP-2.

Although nested vector processing has proved to be a promising approach to programming applications involving irregular data structures on vector machines, it has not yet been widely adopted for programming real-world applications. This is due to the fact that, so far, most efforts have focused on the development of *new* programming languages supporting nested data parallelism. Adopting a new programming language, however, has proved to be very costly and difficult throughout the history of computing. Moreover, special libraries that realise basic vector operations are being developed specifically to implement these new languages, e.g. CVL [7] for NESL. These libraries are usually machine-dependent, often consisting of a number of vector operations implemented in assembler language to obtain the desired efficiency—see, for example, the CRAY Y-MP implementation of CVL. Such a machine-dependent implementation technique constitutes another obstacle to the adoption of nested vector processing.

The goal of the research presented here is not only to integrate the concept of nested vector processing into some imperative language, but to extend Fortran 90 [2], the most widely used data-parallel programming language. The resulting extension is called Fortran 90V. Previous work [11] has integrated nested data parallelism into C, but this has been partially unsatisfactory owing to the omnipresence of pointers in C. The availability of flat vector-based data-parallel operations and allocatable arrays in Fortran, together with the scheduled introduction of pure functions in the next revision, makes Fortran the more attractive candidate for such an extension. Another original aspect of our work on Fortran 90V, with respect to other languages supporting nested data parallelism, is its translation into plain Fortran 90, exploiting the intrinsic data-parallel operations, and thus making use of existing vectorisation technology—this avoids the use of specialized, machine-dependent vector libraries such as CVLand makes Fortran 90V available on all machines supporting Fortran 90. This should make Fortran 90V an ideal tool for nested vector processing. The overall goal of this project is twofold: first, we want to apply the advanced technology of nested vector processing, developed in an academic context, to broaden the application range of the standard language Fortran 90; second, by thus enhancing Fortran, we seek to greatly enlarge vector machines' range of applications. Furthermore, our work also provides the basis for a portable implementation of Fortran 90V on multi-scalar and multi-vector processor machines by employing a standard message-passing library such as MPI [1] in the generated Fortran code.

Section 2 illustrates the general concept of nested data-parallel programming, taking NESL as an example. Section 3 introduces Fortran 90V as a new version of Fortran 90 extended by nested data parallelism. Section 4 presents programming examples are presented for some irregular applications, and Section 5 draws some conclusions.

2 Nested Data-Parallel Programming

NESL, the best known nested data-parallel programming language, is a strongly typed functional language in which parallelism is expressed exclusively by means of the built-in *vector* data type. Most importantly, vectors may be nested, i.e. a vector may itself contain vectors as elements. The predefined operations on vectors consist of a set of primitive parallel functions, such as `permute`, `sum`, `plus_scan`, etc. as well as a generic vector-processing construct, the *apply-to-each*, which allows arbitrary computations to be applied simultaneously to all elements of a vector. An apply-to-each consists of a body, bindings and an optional filter. In the following example, the body `sum (v)` is evaluated for all subvectors bound to `v`. An optional filter expression can restrict the values for which the body is evaluated. Hence, the expression

```
{sum (v) : v in [[2,6], [7,4,7], [6]]}
```

is evaluated to `[8, 18, 6]`.

An apply-to-each (realising the outer parallelism) can have a body which itself specifies a parallel computation (the inner parallelism): either by calling a parallel primitive function, as in the example above, or by employing a second (nested) apply-to-each, which may possibly be hidden within a user-defined function called from the body.

The central idea in the implementation of nested data parallelism on vector computers is the use of the flattening technique, which transforms nested parallel computations into equivalent flat computations while preserving the degree of parallelism. Nested vectors are represented by flat ones as follows: an n-fold nested vector is divided into a flat vector, containing the data of the leaves of the nested structure, the so-called *value vector*, and n flat vectors, called *segment descriptors*. Each segment descriptor contains the lengths of the subvectors of one level, thus describing the nesting structure on that level of the nested vector.

For example, the nested vector `[[[1, 2]], [[3], [4, 5, 6]]]` is represented by the following two segment descriptors and a value vector:

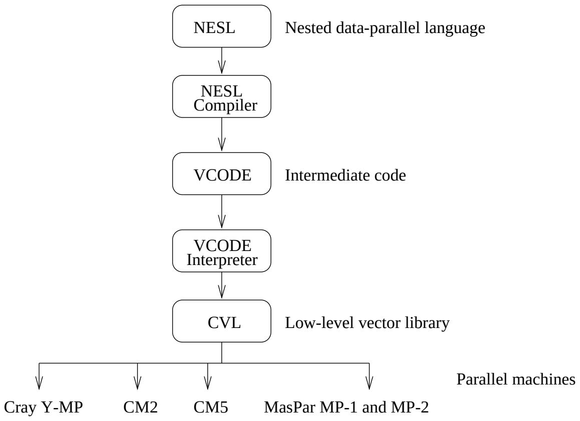
value vector : [$\underline{1,\ 2}$, $\underline{3}$, $\underline{4,\ 5,\ 6}$]
seg. desc. (inner) : [$\underline{2}$, $\underline{1,\ 3}$]
seg. desc. (outer) : [1, 2]

The essence of the flattening transformation is the elimination of the apply-to-each constructs. This is achieved by replacing scalar operations in the body with elementwise vector operations and by replacing vector operations with segmented vector operations (segmented vector operations are described in detail in [4]).

When transforming, for instance, the expression `{x + y :  x in xs; y in ys}`, the scalar body expression `x + y` is *lifted* to the vector expression `plus_vec (xs, ys)` (performing an elementwise addition of two vectors), which is then taken as a substitute for the whole apply-to-each.

A more difficult problem is the treatment of user-defined functions occurring in the body of an apply-to-each. In essence, a vectorised version of such a function has to be generated by the compiler from the scalar version provided by the programmer. This is achieved by transforming the body of the function in the same way as the body of apply-to-each constructs. As a result, we obtain two versions for each user-defined function (used in an apply-to-each): the original version provided by the user, and a vectorised version generated by the transformation. Detailed descriptions of this transformation are available elsewhere [15,20].

In its current implementation, NESL is compiled into a byte code called VCODE [3], which is interpreted using CVL [7], a C-callable library consisting of a number of low-level vector operations. An overview over the NESL system, slightly adapted from [8], is given below:



The CVL library offers two classes of operations: those corresponding to the primitive parallel operations of NESL, such as `permute` or `sum`; and elementwise functions, which apply operations such as addition or multiplication to all elements of an argument vector. All the primitive parallel operations have a segmented version, too, which takes nested vectors as arguments—represented by the corresponding value vector and segment descriptor.

While NESL has convincingly demonstrated the usefulness of nested data parallelism [6], it has three main disadvantages when applied to real-world problems. Because it is a new language and because of its lack of interoperability with other systems, it induces overhead for programmers of application software, who have to learn it and reimplement application-specific libraries accordingly. Secondly, it uses a specialised vector library (CVL) that is partially implemented in assembler code (e.g. on the CRAY Y-MP) and results in considerable overhead when porting the system to new platforms. Thirdly, expressions within the body of an apply-to-each consisting of a number of primitive operations, e.g. `a+b*c-d`, are realised by distinct calls to CVL functions, resulting in superfluous loads and stores of intermediate values and diminishing data locality. Furthermore, CVL implements a very simple load-balancing mechanism and has a naive data-distribution policy which can lead to communication overhead and large memory requirements, especially on distributed-memory machines. Though this makes CVL unsuitable for some applications, the latter problem could be remedied by redesigning the library.

3 Practical Nested Data-Parallel Programming: Fortran 90V

We propose Fortran 90V (F90V) as an extension of the Fortran 90 standard (F90) that allows the use of nested array types. The extension is conservative in the sense that it merely relaxes a restriction on the already available array type of F90—we generalise the multidimensional rectangular array structures to arbitrarily nested structures. The intrinsic array functions of F90 are generalised accordingly. Thus, F90V facilitates the reuse of existing F90 source code and makes it easier for the user to adopt nested vector processing.

3.1 Introducing Irregular Structures

In standard F90, all arrays are rectangular, i.e. all array sections in one dimension must be of the same length. F90V supports *irregular* arrays on which no such restriction is imposed. Such arrays are declared in the same way as standard Fortran arrays, but they are tagged with the special attribute `IRREGULAR`. This reflects the fact that subarrays may be of different extent. The declaration

```
REAL, DIMENSION (:,:), IRREGULAR :: nv
```

declares an irregular array variable `nv`, corresponding to a vector of vectors in NESL. The rank of an irregular array is defined like that of a regular array, namely, as the number of dimensions. We can assign a value to `nv`, for example, by using the constructor for irregular arrays:

```
nv = (<(< 2, 6 >), (< 7, 4, 7 >), (< 6 >)>)
```

Unlike F90 array constructors, the syntactic structure of the constructor is relevant and can be used to construct irregular arrays of ranks other than one.

In F90, the shape of an array of rank n consists of n integer values specifying the size of the array in each dimension. An irregular array can have differents extents for every subarray in every dimension. Thus, there is not only *one* extent value, but as many as there are subarrays in that dimension. All these length values are gathered together in one rank-1 regular array. To specify which length values belong to which dimension, there is a dimension descriptor containing the number of length values for every dimension, in consecutive order. The two arrays form the shape for an irregular array and are combined into a single object of type `ISHAPE` (*irregular* `SHAPE`), which is defined as follows.

```
TYPE ISHAPE
  INTEGER, DIMENSION (:), ALLOCATABLE :: dims
  INTEGER, DIMENSION (:), ALLOCATABLE :: lens
END TYPE ISHAPE
```

The component `dims` is the number of length values, and `lens` contains all the length values themselves. The size of `dims` is equal to the rank of an irregular array. For example:

```
INTEGER, DIMENSION(:,:,:), IRREGULAR :: A
A =                                     &
(<(<(<1,2,3>), (<>)                 >),&
  (<                                >),&
  (<(<4, 5>),  (<6,7>), (<8,9,10,11>)>),&
  (<(<12,13>)                       >))>)
```

The shape of the irregular rank-3 array `A` can be retrieved with the extended intrinsic function `SHAPE`:

```
TYPE(ISHAPE) :: sh
sh = SHAPE (A)
```

we now have `sh%dims = (/6,4,1/)` and `sh%lens = (/3,0,2,2,4,2,2,0,3,1,4/)`.

The current Fortran standard does not permit components of derived types to carry the `ALLOCATABLE` attribute. Instead, pointers have to be used for dynamic memory management. This restriction has already been recognzied as a mistake and will be corrected in future versions of the Fortran standard [16]. Just as allocated arrays are deallocated on exiting a procedure, the components of an `ISHAPE` will most likely also be freed automatically. This will greatly simplify the memory management of irregular shapes. Otherwise, pointers and explicit deallocation would have to be used by the programmer.

A new shape can be attached to an irregular array using the intrinsic function `RESHAPE`. Both irregular and standard shapes can be used to reshape an irregular array. If a standard shape is provided, the result becomes a regular array. The `PAD` and `ORDER` parameters can be used when reshaping irregular arrays to supply a valid number of array elements.

A standard array can be converted into an irregular array using `RESHAPE` by passing an irregular shape. A regular shape can be converted into an analogous irregular shape using the new intrinsic function `RELAX`:

```
TYPE(ISHAPE) :: is
is = RELAX ((/2,4/))
```

then `is = ISHAPE((/4,1/),(/2,2,2,2,4/))`. In this example, a regular shape describing a two-dimensional array with two rows and four columns is converted into an irregular shape. This irregular shape has four subarrays in the second dimension, specified by the last value 4 in the `lens` part. All subarrays have the same extent 2, given by the value 2 repeated four times in the `len` field. This reflects the regular pattern of the standard shape.

Indexing and Subarrays. In F90, single array elements or array sections can be addressed via subscript lists containing an index or an index set for every array dimension. The same is true for irregular arrays in F90V, too. To subselect parts of an irregular array, valid index values for every dimension can be specified. An irregular array of rank n can be thought of as consisting of a number of (irregular) *subarrays* of rank $n-1$. The nth subarray of an irregular array can be retrieved using the subscript values `(:, ... ,:,n)`. A subarray is conceptually equal to a subvector in NESL. Irregular arrays of rank-1 have scalar values as their "subarrays".

Apply-To-Each. In the same way that F90 provides elementwise extensions of the intrinsic scalar functions, such as `+` or `*`, on regular arrays by means of overloading, F90V extends this overloading to irregular arrays. In addition, F90V introduces its variant of the apply-to-each, which has the general form

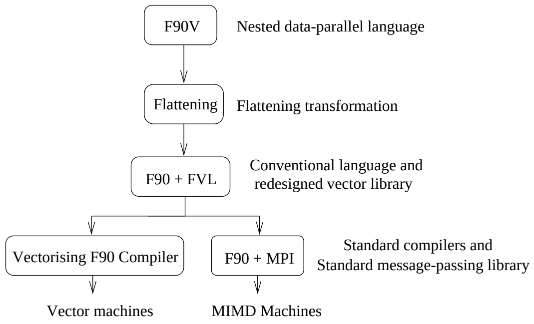
$$\texttt{(<}\ \langle\text{expression}\rangle\ \texttt{:}\ \langle\text{bindings}\rangle\ \texttt{|}\ \langle\text{filter}\rangle\ \texttt{>)}$$

where the three parts play the same role as in NESL, ⟨filter⟩ being optional. (Note that we use the same delimiters as for the irregular array constructors.) A binding has the form ⟨var⟩ = ⟨expr⟩, multiple bindings being separated by commas. A binding specifies that the expression in the body is to be evaluated for every subarray, named by ⟨var⟩, of the irregular array expression ⟨expr⟩. To get a better idea of the construct we should note that for two vectors `AS` and `BS`, `AS + BS` produces the same result as `(< A + B : A = AS, B = BS >)`.

Within an apply-to-each, arbitrary Fortran expressions can be used, in particular those containing user-defined functions. Since side effects endanger semantic properties in a parallel execution, the functions allowed in an apply-to-each are restricted to *pure functions*, i.e. functions with only very restricted side effects. Pure procedures have been introduced in High-Performance Fortran (HPF) [21] and will become part of standard Fortran in its next revision. Note that the apply-to-each can be seen as a generalisation of the `FORALL` construct of HPF.

3.2 Outline of the Implementation

Most F90 compilers on vector machines, such as the Fujitsu VX series and the CRAY Y-MP, vectorise the intrinsic operations on arrays. Moreover, F90 supports the dynamic allocation and memory management of arrays. These features provide the necessary support for a high-level implementation of the nested data parallelism of F90V. We map an F90V program into plain F90 plus a library that can also be realised entirely in F90 itself. Such a high-level implementation is easier to maintain and can utilize the wealth of optimisations available in a vectorising compiler. An overview of the F90V system is given below:



The flattening transformation maps a nested data-parallel F90V program into flat data parallel code by translating the body of apply-to-each constructs as described in the previous section. Irregular arrays with arbitrary nesting depth can be flattened to once-segmented vectors which can be passed to segmented vector operations. The flattened nesting structure can be subsequently reattached to the arrays. Both the flattening and reattachment of the nesting information can be accomplished in constant time. Segmented array operations and other functions that are not provided as intrinsic functions in F90 are realised in a Fortran Vector Library (FVL). FVL itself is implemented in F90 using vectorisable algorithms [10, 12] for the segmented vector operations, which are crucial for the implementation of nested computations. Hence, the efficient implementation of FVL will greatly benefit from the vectorising compilers available on vector machines. Segmented operations can be efficiently implemented, i.e. with the same asymptotic complexity as their unsegmented versions, on scalar parallel machines with distributed memory as well [4]. This approach enhances portability to a wide variety of parallel machines, so that any machine with a F90 compiler has immediate access to F90V.

4 Examplary Applications

Irregular problems have attracted a great deal of interest in scientific computing. Sparse linear systems, for example, are of great importance in engineering computations [18]. It is therefore essential to operate efficiently on irregular and sparse data structures. However, such structures do not lend themselves to simple representation on conventional vector machines—irregular data representations can break the pipeline. This section shows that in F90V such structures can be represented clearly and efficiently.

4.1 Sparse-Matrix Vector Multiplication

A first simple example illustrates an F90V algorithm for sparse-matrix vector multiplication. A sparse matrix can be represented by using an irregular array containing the non-zero values together with their column indices in the matrix. For example, the matrix

$$\begin{pmatrix} 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 6.0 & 8.0 \\ 2.0 & 0.0 & 0.0 & 0.0 \\ 3.0 & 0.0 & 7.0 & 0.0 \end{pmatrix}$$

is represented by

```
(<(< Elem(1.0, 2)                >),  &
  (< Elem(6.0, 3), Elem(8.0, 4) >),  &
  (< Elem(2.0, 1)                >),  &
  (< Elem(3.0, 1), Elem(7.0, 3) >)>)
```

provided that the data type `Elem` is defined as

```
TYPE Elem
  REAL    val
  INTEGER col
END TYPE Elem
```

The function `sparseMVM` takes as its arguments a sparse matrix and a vector, and returns as its result their product.

```
FUNCTION sparseMVM (mat, vec)
  TYPE(Elem), DIMENSION (:,:), INTENT (IN) :: mat
  REAL, DIMENSION (:), INTENT (IN)   :: vec
  REAL, DIMENSION (:),               :: sparseMVM

  IRREGULAR :: mat, vec, sparseMVM

  sparseMVM =                                      &
  (< sum((< e%val*vec(e%col) : e=r >)) : r=mat >)
END FUNCTION sparseMVM
```

For each row r of the sparse matrix mat, the non-zero elements of that row are multiplied with the corresponding elements of the vector vec—the latter being obtained by indexing vec at e%col—and the resulting products are summed up.

The flattening process transforms the body of sparseMVM into the following F90+FVL code:

```
! FVL function: unsegmented backward permutation
PERMUTE:
CALL fvl_backpermute(vecvals,vec,mat_values%col)

! Fortran 90 array operation
PRODUCTS:
prods = mat_values%val * vecvals

! FVL function: segmented reduction with +
SUM:
CALL fvl_segmented_sum(result,prods,mat_segd1)

sparseMVM = result
```

Here, mat_values is the value vector of mat and mat_segd1 its segment descriptor, which is a part of the internal representation of the array's shape.

When executing the multiplication

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 6 & 8 \\ 2 & 0 & 0 & 0 \\ 3 & 0 & 7 & 0 \end{pmatrix} \begin{pmatrix} 9 \\ 1 \\ 4 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 \\ 40 \\ 18 \\ 55 \end{pmatrix}$$

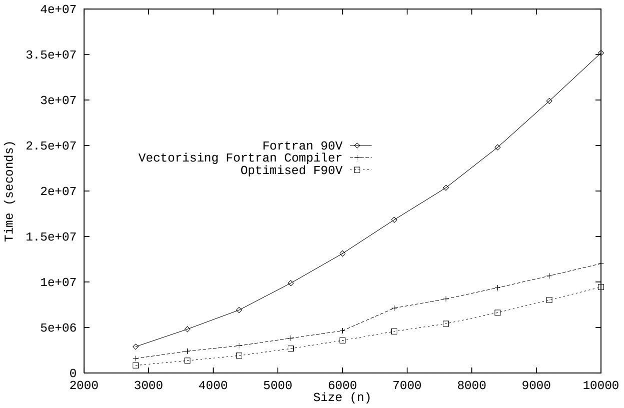
we have the following valuation of the variables on entering the function sparseMVM:

```
mat_values%val   :  (/1, 6, 8, 2, 3, 7/)
mat_values%col   :  (/2, 3, 4, 1, 1, 3/)
mat_segd1        :  (/1, 2, 1, 2/)
vec              :  (/9, 1, 4, 2/)
```

Tracing the execution of the code, we obtain the following results:

Label	Variable	Value
PERMUTE	vecvals	(/1, 4, 2, 9, 9, 4/)
PRODUCTS	prods	(/1, 24, 16, 18, 27, 28/)
SUM	result	(/1, 40, 18, 55/)

The following graph shows performance measurements of the sparse matrix vector multiplication, where the matrices had dimensions $n \times n$ with a density ratio of 1%:



All programs were run on a CRAY EL94 with two vector processors and 256 MB memory. The F90V program is slower than the vectorised version of a standard Fortran program. This is because each function call to the FVL has to store the result back into the memory and call another one, which will break the pipleline. If we optimise away the function calls and hence do not break the pipeline anymore, the performance becomes better than that of standard Fortran. The standard Fortran90 algorithm is shown below:

```
SUBROUTINE SMVM (ans, mat, vec, chunkSize)
TYPE(Elem), DIMENSION(*) :: mat
INTEGER, DIMENSION(*) :: ans, vec, chunkSize
INTEGER i, j, start, end
 start = 1
 DO i = 1, SIZE(chunkSize)
   end = start + chunkSize(i) - 1
   ans(i) = sum(mat%val(start:end) * &
            vec(mat%col(start:end)))
   start = end + 1
 END DO
END SUBROUTINE
```

4.2 The Quickhull Algorithm

A fundamental problem in computational geometry, for which different parallel solutions have been proposed, is the convex-hull problem [14]. In F90V, we can express a variation of the quickhull algorithm [6, 19] quite elegantly. This example illustrates the use of nested data parallelism for divide-and-conquer algorithms, and its code is given in Figure 1. The code uses values of type COMPLEX to represent points on a two-dimensional plane. The calculation is based on the recursive function hsplit, which finds all points lying on the convex hull clockwise between the two points p1 and p2,

```
FUNCTION quickhull (points)
  COMPLEX, DIMENSION (:), IRREGULAR, INTENT (IN) :: points
  COMPLEX, DIMENSION (:), IRREGULAR              :: quickhull
  REAL,    DIMENSION (:), IRREGULAR              :: xvals
  COMPLEX                                           maxx, minx

  xvals = (< real(p) : p = points >)
  minx = points(minloc (xvals))
  maxx = points(maxloc (xvals))
  quickhull = flatten ((< hsplit (points, m, n) : m = (< minx, maxx>), n = (< maxx, minx>) >))
END FUNCTION quickhull

PURE RECURSIVE FUNCTION hsplit (points, p1, p2) RESULT (hull)
  COMPLEX, DIMENSION (:), INTENT (IN), IRREGULAR :: points
  COMPLEX,                INTENT (IN)            :: p1, p2
  REAL,    DIMENSION (:),            IRREGULAR :: cross
  COMPLEX, DIMENSION (:),            IRREGULAR :: packed
  COMPLEX                                           pm

  cross =  (< crossproduct (p, p1, p2) : p = points >)
  packed = (< p : p = points, c = cross : c >= 0.0 >)
  IF (size (packed) < 2) THEN
    hull = (< p1 >) // packed
  ELSE
    pm = points(maxloc (cross))
    hull = flatten ((< hsplit (packed, end1, end2) : end1 = (< p1, pm >), end2 = (< pm, p2 >) >))
  END IF
END FUNCTION hsplit

PURE FUNCTION crossproduct (point, start, end)
  TYPE(point), INTENT (IN) :: point, start, end
  TYPE(point)              :: start0, end0
  REAL                        distance

  start0 = start - point
  end0 = end - point
  crossproduct = real(start0) * aimag(end0) - aimag(start0) * real(end0)
END FUNCTION crossproduct
```

Figure 1: The Quickhull algorithm in Fortran 90V.

where the latter are already known to belong to the hull. The routine first removes all points lying below the line that connects p1 with p2. Then, the point with the maximum distance to the line is computed. This point is part of the hull. Now, there are two new pairs of points (p1 and the maximum, as well as the maximum and p2). The function hsplit then calls itself in parallel for both pairs using an apply-to-each. The function quickhull determines the two points with the minimum and the maximum x coordinates. These points must belong to the convex hull and serve as initial points for hsplit. The F90 intrinsic functions MINLOC and MAXLOC are supplied for irregular arrays. As with strings, irregular arrays of arbitrary rank can be concatenated by the operation //.

F90V Intrinsic Functions. The FLATTEN routine decreases the rank of an irregular array by a special kind of reshaping: the highest dimension is eliminated by collecting all the elements extending in this dimension in array element order and placing them in the last-but-one dimension—in other words, the topmost nesting structure is removed.

In NESL, there are a number of intrinsic functions that have proved to be useful for expressing parallel algorithms on a high level of abstraction. Some of these functions are provided in F90V, too. Most of them are "subarray oriented" rather than element-oriented, as most F90 intrinsic operations are.

PLUS_SCAN takes a rank-1 array of numerical type and returns all partial prefix sums. The scan operation is also provided for the operations MAX, MIN, AND, OR and MULT. Prefix sums can be used in a wide range of parallel algorithms [5]. PERMUTE can be used to permute the subarrays of an irregular array of arbitrary nesting depth by an index array. An irregular array of rank n can be replicated l times to give an array of rank $n+1$ with the extended primitive `SPREAD`. `PARTITION` is the inverse operation of `FLATTEN`. It subdivides an irregular array into parts that become the subarrays of a new dimension, thus yielding an array of increased rank. The parts are specified in a rank-1 standard array of lengths, i.e. by a segment descriptor. All theses functions are pure functions and can be used in parallel in an apply-to-each.

5 Conclusion

The design of F90V has been finalised, and we are currently completing the implementation of the FVL library. The FVL is implemented entirely in Fortran 90 in order to increase portability, albeit at the cost of performance. We can expect native implementations of the FVL to be significantly faster.

The integration of nested data parallelism into Fortran 90, together with the outlined highly portable implementation technology, should enable the average programmer to use nested vector computations, facilitating the implementation of applications based on irregular data structures on vector computers and other parallel machines supporting Fortran 90.

Acknowledgements. We would like to thank Phil Bacon for his help in preparing this paper. Peter Au's work is funded by the Croucher Foundation. Gabriele Keller's work is being supported by a PhD scholarship from the German Research Foundation (DFG). We greatefully acknowledge Fujitsu's support in providing us access to the vector and parallel machines in the Imperial College / Fujitsu Parallel Computing Research Centre (IFPC) and Fujitsu European Centre for Information Technology (FECIT).

References

[1] *Message Passing Interface Forum. MPI: A Messsage-Passing Interface Standard*, May 1994.

[2] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran 90 Handbook Complete ANSI/ ISO Reference.* McGraw-Hill, 1992.

[3] G. E. Belloch, S. Chaterjee, J. Sipelstein, and M. Zagha. *VCODE Reference Manual (Version 2.0).* Carnegie Mellon University, Feb 1994.

[4] G. E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT, 1990.

[5] G. E. Blelloch. Prefix Sums and Their Applications. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, pages 35–60. Morgan Kaufman Publishers, 1993.

[6] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[7] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, M. Reid-Miller, J. Sipelstein, and M. Zagha. CVL: A C vector library manual (Version 2). Technical Report CMU-CS-93-114, Carnegie Mellon University, Feb 1993.

[8] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, Apr. 1994.

[9] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, and M. Zagha. *NESL User's Manual (Version 3.1).* Carnegie Mellon University, Sept 1995.

[10] G. E. Blelloch, M. A. Heroux, and M. Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. Technical Report CMU-CS-93-173, Carnegie Mellon University, Aug. 1993.

[11] M. M. T. Chakravarty, F. W. Schröer, and M. Simons. V—Nested parallelism in C. In W. K. Giloi, S. Jähnichen, and B. D. Shriver, editors, *Programming Models for Massively Parallel Computers*, pages 167—174. IEEE Computer Society, 1995.

[12] S. Chatterjee, G. E. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Proceedings, Supercomputing '90*, Nov 1990.

[13] R. Faith, L. Nyland, D. Palmer, and J. Prins. The Proteus ns grammer. Technical report, UNC-CH, 1994.

[14] J. JáJá. *An Introduction to Parallel Algorithms.* Addison Wesley, 1992.

[15] G. Keller and M. Simons. A calculational approach to flattening nested data parallelism in functional languages. In J. Jaffar, editor, *The 1996 Asian Computing Science Conference*, Lecture Notes in Computer Science. Springer Verlag, 1996.

[16] M. Metcalf and J. Reid. *Fortran 90/ 95 Explained.* Oxford Scienece Publications, 1996.

[17] D. W. Palmer, J. F. Prins, and S. Westfold. Work-efficient nested data-parallelism. In *Prcoeedings of the Fifth Symposium in the Frontiers of Massively*, pages 186–193. IEEE, 1995.

[18] V. Pan. Parallel solution of sparse linear and path systems. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, pages 621–678. Morgan Kaufman Publishers, 1993.

[19] F. Preparata and M. Shamos. *Computational Geometry—An Introduction.* Springer Verlag, 1985.

[20] J. F. Prins and D. W. Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[21] Rice University. *High Performance Fortran Forum. High Performance Fortran Language Specification.*, 1.1 edition, Nov 1994.

[22] G. W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming.* The MIT Press, 1988.