Prototyping Sparse Fortran 90 Array Intrinsics with Standard ML Module System*

Tyng–Ruey Chuang

Institute of Information Science
Academia Sinica
Nankang, Taipei 11529, Taiwan

trc@iis.sinica.edu.tw

Abstract

We report work-in-progress of using the Standard ML module system to prototype sparse Fortran 90 array intrinsic functions. We illustrate type-related issues in a high-level, source-to-source mapping of Fortran 90 array intrinsics to Standard ML. The prototype sparse library is parameterized by array compression schemes. Various issues involved in using Standard ML as a prototyping language are also discussed.

1 Motivation

Implementations of high-level programming languages often rely on run-time library to support built-in language features. For example, garbage collection routines are needed at run-time for languages supporting automatic memory management, and process scheduler for languages support multithreaded execution. These run-time libraries are often written using high-level languages themselves. A major benefit of using high-level languages for writing library code is that it allows rapid experimentations of various design decisions. Sometimes the library is implemented in the source language, in which case the library may become directly accessible to user programs, and may allow user-defined customization or even replacement of the library. Though now one must make sure that the interface to the user-customized library is type-safe. The module system of Standard ML, which allows parameterized and type-checked generation of library units (*i.e.*, structures) by using its functor facility, is an excellent choice when experimenting with library design, even when Standard ML is not the source nor the implementation language of the library.

We report in the paper experience in using Standard ML as a prototyping language for library design. Our source language is Fortran 90, a language supports high-level array computation. Our aim is to implement in Fortran 90 a library to support sparse array computation. (Dense) array computation in Fortran 90 is well supported by array intrinsics (*i.e.*, language built-in procedures,

*This paper is available on-line as technical report TR-IIS-98-002 from the Institute of Information Science, Academia Sinica, via `http://www.iis.sinica.edu.tw`.

details in Section 2). Our goal can be thought of providing a customization of the array intrinsics, but only for sparse arrays. That is, interface to both the original dense array intrinsics and our sparse array intrinsics will be kept the same, and equally accessible to the programmers.

Sparse array computation is a challenging topic because efficient computation depends greatly on the data representations of the sparse arrays (*i.e.*, on how they are compressed), which again depends on the sparsity structures (*i.e.*, the distributions of non–zero elements) of the arrays [2]. Sparsity information, however, may not be available at compile–time. Even if an array's sparsity structure is available at compile–time, its structure may change over time during execution, which makes it more difficult for selecting good compression schemes. In parallel sparse array computation, in addition to compression schemes, one also faces the problem of selecting proper distribution schemes.

For programmers, it will be easier if the languages they use provide run-time support of automatic selection of data representations for sparse arrays. However, current high-level array languages either do not support sparse arrays (as in APL and Fortran 90), or only provide fixed compression schemes (as in MATLAB [5]).

2 Fortran 90 Arrays and Array Intrinsics

Fortran 90 is a much improved language over Fortran 77. It provides rich intrinsic procedures to support whole array operations. These intrinsic procedures for arrays are called *array intrinsics*. Array intrinsics encourage a style of parallel programming known as data parallelism. High Performance Fortran, which is based on Fortran 90, provides further facility for customizable data distribution (*i.e.*, array partition) and processor allocation, and is perhaps the only parallel programming language supported by major computer manufacturers (as well as independent software vendors). Fortran 90 has been gradually accepted to the numerical computing community, as exemplified by the appearance of the books like *Numerical Recipes in Fortran 90* [7].

Each array in Fortran 90 has a *shape*, referring to both its *rank* (the number of dimensions) and *extents* (the lengths along each dimension). The shape of an array is specified by a rank-one integer array whose elements are the extents along each dimension. The *size* of an array is the total number of elements in the array. The rank of an array cannot be larger than 7. Array elements are homogeneous and can be of intrinsic data types (such as `integer`, `real`, `complex`, `logical`, and `character`) or derived data types (*i.e.*, user-defined data types).

Array operations are well supported in Fortran 90 because the language provides the following language features.

Better array notations. For a $n \times n$ real array `a` declared by

```
real, dimension (n, n) :: a
```

Then the notation `a(1, :)` refers to its first row, and `a(:, n)` its last column. Further, `a(1:n:2, 2:n:2)` is the sub-array whose elements each has both an odd row index and an even column index in `a`. These are called *array sections* and can appear at either sides of an assignment.

Elemental intrinsic operations. Most Fortran 90 intrinsic functions (such as `abs`, `+`, *etc.*) are elemental, meaning that the argument can be an array of arbitrary shape as well as a scalar value. For example, the assignment `a = a + a` doubles the value of each element in array `a`. Note that it is required the two array operands to the `+` function are of the same shape. That is, they must be *conformable*. Storage for arrays can be be allocated either at compile-time or run-time. The life-time of run-time allocated arrays can be controlled by programmers by using storage management procedures `allocate` and `deallocate` (which are similar to `malloc` and `free` in C).

Most Fortran 90 intrinsic functions are *generic* as well. For example, `+` can be used for integer number addition and real number addition. In Fortran 90, one can create new, user-defined, generic functions by overloading existing function names or operators. However, Fortran 90 provides no easy facility to define new, user-defined, elemental functions. This is one of the reasons why we use Standard ML to prototype sparse array intrinsics: Standard ML provides a parameterized module facility from which elemental functions can be easily built up.

Transformational array intrinsics. Fortran 90 provides several powerful array transformation procedures, such as the following.

`merge` returns an conformable array whose elements are selected from two source arrays based on a boolean mask array.

`spread` returns an array whose rank is one greater than the input array by duplicating the input array along some given dimension.

`pack` packs an array of arbitrary shape into an rank-one array using a mask array.

`unpack` unpacks an rank-one array into an array of arbitrary shape using a mask array.

`reshape` maintains the Fortran order (*i.e.*, column major) of the elements in the input array but puts them in an array of different shape.

`cshift` returns the result of circularly shifting every one-dimensional sections of the input array.

`eoshift` returns the result of end-of shifting every one-dimensional sections of the input array.

`sum`, `product`, `...` are reduction operations.

etc.

These intrinsics can be used to write concise array expressions. As an example (taken from [7]), the calculation

$$w_i = \sum_{j=1}^{n} |x_i + x_j|, \qquad i = 1, \ldots, n$$

can be realized as

```
real, dimension (n) :: x, w
real, dimension (n, n) :: a
...
a = spread(x, dim=2,ncopies=n) + spread(x, dim=1, ncopies=n)
w = sum(abs(a), dim=1)
```

of which the first expression can be illustrated by

$$a_{i,j} = x_i + x_j = \begin{pmatrix} x_1 & x_1 & x_1 & \cdots \\ x_2 & x_2 & x_2 & \cdots \\ x_3 & x_3 & x_3 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} + \begin{pmatrix} x_1 & x_2 & x_3 & \cdots \\ x_1 & x_2 & x_3 & \cdots \\ x_1 & x_2 & x_3 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

Detailed descriptions of Fortran 90 array intrinsics can be found in standard reference (such as [1]) and are omitted here.

3 Standard ML as a Prototyping Language

Fortran 90 does supports modular programming development. Its module facility allows one to define the interface of an abstract data type without revealing its implementation. Program units can be type-checked at link-time to assure type-safe linkage of imported modules. (Though not necessary every Fortran 90 compiler performs this check. Still, this is a big improvement over C, so to speak.) User-defined functions/procedures can be overloaded to maintain ad hoc polymorphism.

Fortran 90 further supports pointers (again, safer than C), recursive user-defined data types, recursive functions and procedures, passing functions/procedures as arguments (but not returning as results), dynamic storage allocation/deallocation. Combining these language features with its module facility, one can develop very sophisticated user library in Fortran 90.

We are in the process of building in Fortran 90 a library to support sparse array intrinsics [4]. By sparse array intrinsics, we mean that sparse arrays are used in the same ways as (dense) arrays in Fortran 90. That is, convenient notations can be used, elemental intrinsic functions can be applied to sparse arrays, and transformational array intrinsics applies to sparse arrays as well. Being a library, there are certain restrictions on the level of convenience it can achieve. Still, we are able to provide a library that supports the following style of sparse array programming:

```
type (sparse1d_real) :: x, w
type (sparse2d_real) :: a
call bound (x, n)
call bound (w, n)
call bound (a, n, n)
...
a = spread(x, dim=2,ncopies=n) + spread(x, dim=1, ncopies=n)
w = sum(abs(a), dim=1)
```

where we take the example in Section 2 and rewrite it using our sparse library. Notice that functions `spread`, `sum`, `abs`, `+` have been overloaded to support sparse arrays. Also notice that storage allocation (`call bound`) have to be performed manually (since Fortran 90 does not support initialization constructors).

However, we gradually find out that Fortran 90 is not a good language for building sparse array intrinsic library. The main two reasons are the following.

- Although Fortran 90 is a typed language, it is not suitable for programming about types. For a sparse array intrinsic library to fully function, it has to support arrays of every ranks, as well as every applicable intrinsic data types. This means that for the `+` function, one has to define 21 different addition functions (7 ranks, 3 intrinsic data types — `integer`, `real`, `complex`) and overload `+` to the 21 functions. (Let's call this process the elementalization of the `+` operator.) Most of the code in the 21 functions looks the same. Since Fortran 90 is a monomorphic language, one really cannot do much here.

- Fortran 90 does not have automatic storage management. Managing storage for the various compression schemes of sparse arrays becomes tiresome. One major goal of the sparse library is to experiment different kinds of data representations of sparse arrays, and the interactions between them and the various transformational array intrinsics (`reshape`, `cshift`, *etc.*). The burden of manual management of storage, and the bugs so introduced, prevents quick experimentations of new compression schemes.

Standard ML, on the other hand, has a polymorphic type system and supports automatic memory management. These two features along fills the above two gaps in the Fortran 90 language, and allow one to quickly devise new ways of constructing sparse array routines. The parameterized module facility in Standard ML further helps us prototype the sparse library. Functors in Standard ML allow one to specify the construction of a structure (library code) based on the signature (library interface) of other structures or functors. The construction can be applied as needed but the specification can be checked for type correctness in advance. This allows one to automatically generate type-safe sparse library code based on existing sparse library code.

Take the example of elementalization of the `+` operator above. One can achieve this easily in Standard ML by the following. One first defines a signature `INTRINSIC` for intrinsic data type a signature `ELEMENTAL_INTRINSIC` for elementalized intrinsic data type, a signature `BLOCK` for whole array operations. One then specify a functor `Elemental` that accept structures of `INTRINSIC` and `BLOCK` signatures and produce a structure of signature `ELEMENTAL_INTRINSIC`. Finally the various different addition functions can be produced, automatically, by applying the `Elemental` functor to various combinations of `INTRINSIC` and `BLOCK` structures. The following is a sample code segment.

```
signature INTRINSIC =
sig
    type t
    val + : t * t -> t
    val * : t * t -> t

    val ABS : t -> t
    ...
end

signature ELEMENTAL_INTRINSIC =
sig
    type t
```

```
    type 'a array

    val + : t array * t array -> t array
    val * : t array * t array -> t array

    val ABS : t array -> t array
    ...
end

signature BLOCK =
sig
    type 'a array

    val map1: ('a -> 'b) * 'a array -> 'b array
    val map2: ('a * 'b -> 'c) * 'a array * 'b array -> 'c array
    val map3: ('a * 'b * 'c -> 'd) * 'a array * 'b array * 'c array -> 'd array
    val fold: 'a * ('a * 'a -> 'a) * 'a array -> 'a
    ...
end

functor Elemental (structure B: BLOCK; structure I: INTRINSIC):> ELEMENTAL_INTRINSIC
  where type t = I.t
    and type 'a array = 'a B.array
=
struct
    type t = I.t
    type 'a array = 'a B.array

    val op + = fn (A, B) => B.map2 (I.+, A, B)
    val op * = fn (A, B) => B.map2 (I.*, A, B)

    fun ABS A = B.map1 (I.ABS, A)
    ...
end
```

One thing that is supported in Fortran 90 but lacking in Standard ML is the mechanism to simultaneously overload the + operator at the top-level to the various addition functions in the resulting structures. But one can argue that overloading is just syntactic sugaring, and is less important than automatic code generation in library design. More examples of parameterized library code generation is described in the next section.

4 Mapping Fortran 90 Arrays to Standard ML

Our objectives in using Standard ML to prototype a sparse array library is to allow quick experimentations of various data representation schemes for sparse arrays, and to observe their effects on the

implementation of various Fortran 90 array intrinsics (especially for the higher-rank, transformational array intrinsics). We are not aiming at a perfect imitation of the full Fortran 90 array intrinsics in Standard ML. The first priority is to design Standard ML functors that are parameterized by various design decisions in the construction of a sparse library. The other priority is to model type-related properties of Fortran 90 arrays in the prototype. We want to use the prototype to express Fortran 90-style array computation in Standard ML, and to check the array expression (now in Standard ML) at different stages (compile-time or run-time) for conformance to various Fortran 90 rules about array properties.

Let us address the type-related properties first. For example, Fortran 90 check the rank of an array expression at compile-time, so we want the prototype to check this property at compile-time as well. It causes a compile-time error to add a rank-one array to a rank-two array in Fortran 90, so the corresponding array addition expression in Standard ML will cause a static elaboration error. Fortran 90, however, checks whether or not the two arrays are of the same shapes (and same size) only at run-time. Therefore, the prototype should implement this check at run-time too. Syntactic sugaring, such as operator overloading and automatic type coercion, however, will not be addressed in the prototype and will need explicit resolutions using long identifiers (*i.e.*, `structure_name.operator_name`) and type-conversion functions.

Since the rank of an array is an elaboration-time property in the Standard ML prototype, it is only nature to designate 7 structures for arrays with one structure for arrays of each rank. For now, we can call the 7 structures `R1`, `R2`, ..., `R7`. Note that it is necessary to refer to $\texttt{R}_{k-1}$ inside $\texttt{R}_k$ because the type of subscript operation in $\texttt{R}_k$ will refer to the array type in $\texttt{R}_{k-1}$. The 7 structures can have different signatures, but that will lead to code explosion since each of the functors that accept array structures will need seven different versions just to be type-checked. With a little twist, we use a single signature for the 7 structures, and generate the 7 structures by successively iterative applications of an array-constructing functor. This part of the prototype code looks like the following:

```
signature BLOCKS =
sig
    structure B: BLOCK

    include BLOCK

    val \ : 'a B.array * int -> 'a   array
    val ? : 'a   array * int -> 'a B.array
end

functor Block () :> BLOCK     (* Fake scalars as arrays!             *)
  where type 'a array = 'a    (* But let the SML type checker know!! *)
=
struct
    type 'a array = 'a

    fun map1 (f, A)       = f  A
    fun map2 (f, A, B)    = f (A, B)
```

```
    fun map3 (f, A, B, C) = f (A, B, C)
    fun fold (e, f, A)    =    A
    ...
end

functor Blocks (structure Base: BLOCK; structure Indexable: VECTOR):> BLOCKS
  where type 'a B.array = 'a Base.array
=
struct
    structure B = Base

    type 'a array = 'a B.array Indexable.vector

    fun \ (a, n) = R.tabulate (n, fn i => a)
    fun ? (A, i) = R.sub (A, i)
    infix  8 ?

    fun map1 (f, A) = R.tabulate (R.length A, fn i => B.map1 (f, A?i))
    ...
end

structure R0 = Block  ()
structure R1 = Blocks (structure Base=R0; structure Indexable = ...)
...
structure R7 = Blocks (structure Base=R6; structure Indexable = ...)
```

A little complication can occur because the ranks of the resulting arrays of some Fortran 90 array intrinsic function may actually depend on the sizes of its argument arrays. For example, for any array A of size 6, `reshape (A, (/2, 3/))` will return a rank-two array of shape `(/2, 3/)`. (Note: `(/2, /3)` is the Fortran 90 notation for a constant, rank-one integer array of elements 2 and 3.) While `reshape (A, (/6/))` will return a rank-one array. Recall that size is not a static property of a Fortran 90 array. Fortran 90 solves this problem by requiring those argument arrays to be of constant sizes (*e.g.*, `(/2, 3/)` has constant size 2, and `(/6/)` has constant size 1), hence allows static rank-checking of the array expression. However, it is difficult to specify and check the constant size requirement in our Standard ML prototype because the prototype is a library, not a compiler.

Our workaround is to provide 7 different reshape functions, with one for each possible size of the second argument:

```
val reshape1: 'a array * int -> 'a R1.array
val reshape2: 'a array * (int * int) -> 'a R2.array
val reshape3: 'a array * (int * int * int) -> 'a R3.array
val reshape4: 'a array * (int * int * int * int) -> 'a R4.array
val reshape5: 'a array * (int * int * int * int * int) -> 'a R5.array
val reshape6: 'a array * (int * int * int * int * int * int) -> 'a R6.array
val reshape7: 'a array * (int * int * int * int * int * int * int) -> 'a R7.array
```

Note that the second array argument is now realized as a tuple in the prototype.

Now let us return to the issues of using parameterized Standard ML functors to model design decisions when building a sparse library. Perhaps the most important decision is about how the arrays are compressed. Currently we are only experimenting various compression schemes for vectors (*i.e.*, rank-one arrays) and progressively compress an array, rank after rank. (This rules out the possibility of compressing higher-rank arrays by k-way search trees with whole array indices as keys, for example.) However, in our design, the compression at each rank may not be the same. All compression schemes are currently specified by the VECTOR signature (as from the SML/NJ Basis Library), but need different implementations. The part of the code looks like:

```
signature F90ARRAY =
sig
    structure R1: ARRAYS
    structure R2: ARRAYS
    structure R3: ARRAYS
    structure R4: ARRAYS
    structure R5: ARRAYS
    structure R6: ARRAYS
    structure R7: ARRAYS

    sharing type R2.B.array = R1.array
        and type R3.B.array = R2.array
        and type R4.B.array = R3.array
        and type R5.B.array = R4.array
        and type R6.B.array = R5.array
        and type R7.B.array = R6.array
    ...

    val \ : 'a * int -> 'a R1.array            (*   array constructor   *)
    val \\ : 'a * (int * int) -> 'a R2.array   (*   array constructor   *)
    ...
    val \\\\\\\ : 'a * (int * int * int *int * int * int * int) -> 'a R7.array

    val ? : 'a R1.array * int -> 'a            (*   array indexing   *)
    val ?? : 'a R2.array * (int * int) -> 'a   (*   array indexing   *)
    ...
    val ??????? : 'a R7.array * (int * int * int *int * int * int * int) -> 'a
end

... ... ...

functor F90Array (structure Indexable1: VECTOR;
                  structure Indexable2: VECTOR;
                  structure Indexable3: VECTOR;
                  structure Indexable4: VECTOR;
                  structure Indexable5: VECTOR;
                  structure Indexable6: VECTOR;
```

```
                  structure Indexable7: VECTOR;
                  structure Integer:    INTRINSIC   where type t = int;
                  structure Real:       INTRINSIC   where type t = real;
                  structure Logical:    INTRINSIC   where type t = bool;
                  functor   Block:      fBLOCK;
                  functor   Blocks:     fBLOCKS;
                  functor   Elemental:  fELEMENTAL;
                  functor   Arrays:     fARRAYS) :> F90ARRAY
where type 'a R1.B.array = 'a
=
struct
local
    structure B0 = Block  ()
    structure B1 = Blocks (structure Base=B0; structure Indexable = Indexable1)
    structure B2 = Blocks (structure Base=B1; structure Indexable = Indexable2)
    ...
    structure B7 = Blocks (structure Base=B6; structure Indexable = Indexable7)
in
    structure R1 = Arrays (structure Blocks = B1; structure Integer = Integer;
                           structure Real = Real; structure Logical = Logical;
                           functor Elemental = Elemental)
    structure R2 = Arrays (structure Blocks = B2; structure Integer = Integer;
                           structure Real = Real; structure Logical = Logical;
                           functor Elemental = Elemental)
    ...
    structure R7 = Arrays (structure Blocks = B7; structure Integer = Integer;
                           structure Real = Real; structure Logical = Logical;
                           functor Elemental = Elemental)
    ...
    infix 8 \ \\ \\\ \\\\ \\\\\ \\\\\\ \\\\\\\
    infix 8 ? ?? ??? ???? ????? ?????? ???????

    val op \ = R1.\
    fun op \\ (x, (a,b)) = R2.\ (x \ b, a)
    ...
    fun op \\\\\\\ (x, (a,b,c,d,e,f,g)) = R7.\ (x \\\\\\ (b,c,d,e,f,g), a)

    val op ? = R1.?
    fun op ?? (A, (a,b)) = R2.? (A,a) ? b
    ...
    fun op ??????? (A, (a,b,c,d,e,f,g)) = R7.? (A,a) ?????? (b,c,d,e,f,g)
end
end
```

In the above, signature `ARRAYS` is similar to `BLOCKS` but with more detailed specification, and `Arrays` a functor that produces a structure of `ARRAYS` signature. Notice the series of `sharing type` constraints

in signature `F90ARRAY`, and the additional `where type` assurance for the resulting `F90ARRAY` signature for functor `F90Array`. Chaining the 7 array structures in this way assures that arrays of different ranks admit certain kinds of interoperability between them (*i.e.*, the lower-rank sub-arrays of a higher-rank array have the same implementations of lower-rank arrays.)

Also notice in the above that block $\texttt{B}_k$ always use compression scheme $\texttt{Indexable}_k$. This need not be the case. We can have $\{\texttt{B}_i\}_i$ compressed using a permutation of the compression schemes $\{\texttt{Indexable}_i\}_i$. However, in order to specify this we need a functor that is parameterized by values from a set of constants, and allows conditional elaboration in the functor body. (A permutation is an array of constants $1, 2, \ldots, n$, though not necessarily in the exact order). The current Standard ML module system does not support this.

5 On-going and Related Work

We discuss here several pragmatic issues when using Standard ML as a prototyping language. In this paper, Standard ML is a prototyping language in the sense that an expression in the source language (Fortran 90 in this case) has a corresponding expression in Standard ML; and the operational semantics of the source expression can be observed by evaluating the corresponding Standard ML expression. In our usage of Standard ML as a prototype language, the smaller the semantic difference between the source language and Standard ML, the easier the prototyping process. Our effort is greatly helped by the facts that both Fortran 90 and Standard ML are strict and statically typed languages, and that Fortran 90 array expressions are often written in a functional style.

One problem with this style of prototyping is that, though operational semantics of the two expressions have a direct mapping, their performance characteristics may be quite different. This effects one's judgment of whether or not one has reached a good prototype because performance of its counterpart in the source language may be much slower/faster than the prototype. A similar problem occurs when one wants to provide an implementation as described in the prototype using the source language. How should the translation be done? Manual mapping is tedious and error-prone. Worst yet, manual translation may be difficult since the prototyping language (Standard ML) often is of much "higher-level" than the source language (Fortran 90). In general, one may have to develop an automatic translator from the prototyping language to the source language. This task can be greatly reduced if the prototyping language has an "open" implementation such that user-defined functions can be hooked to the compiler/interpreter. We are evaluating the feasibility of using the "visible compiler" of Standard ML of New Jersey, Version 109, to help the translation back to Fortran 90.

Chen and Cowie used Standard ML to prototype Fortran 90 compilers for massively parallel machines [3]. However, our perspective of a prototyping process is quite different from theirs. They basically used Standard ML as a high-level implementation language for compiling a program from the source language (Fortran 90) to a program in the target language (assembly code). We use Standard ML as both the implementation language and target language, and the "compilation process" is a high-level source-to-source translation to be performed by the users. The eventual goal of our prototype, which is not yet completed, will be to produce a library in the source language. In short, we design libraries while they implement compilers.

Leroy implemented a Standard ML-like module system as a functor parameterized by the base

language and its associated type-checking functions [6]. The implementation is in Caml (Special Light), a Standard ML-like language. One can say that he provided a prototype for a Standard ML-like module system in Caml, while we provided a prototype for sparse, Fortran 90-like array intrinsics in Standard ML. The two efforts share a similar perspective of the prototyping process and both use Standard ML-like languages.

References

[1] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brain T. Smith, and Jerrold L. Wagener. *Fortran 90 Handbook*. Intertext Publications/McGraw–Hill Inc., 1992.

[2] Aart J. C. Bik and Harry A. G. Wijshoff. Automatic data structure selection and transformation for sparse matrix computations. *IEEE Transations on Parallel and Distributed Systems*, 7(2):109–126, February 1996.

[3] Marina Chen and James Cowie. Prototyping fortran-90 compilers for massively parallel machines. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 94–105. San Francisco, California, USA, June 1992. ACM Press.

[4] Tyng–Ruey Chuang, Rong–Guey Chang, and Jenq Kuen Lee. Sampling and analytical techniques for data distribution of parallel sparse computation. In *Eighth SIAM Conference on Parallel Processing for Scientific Computing*. Minneapolis, Minnestota, USA, March 1997. 8 pages. SIAM Press.

[5] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333—356, January 1992.

[6] Xavier Leroy. A modular module system. Research Report 2866, INRIA, France, April 1996.

[7] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in Fortran 90: The Art of* Parallel *Scientific Computing*. Cambridge University Press, 1996.