RC 21090 (94350) 01/30/98
Computer Science/Mathematics

IBM Research Report

A Case Study of Fortran in Computational Science and Engineering

José E. Moreira, Samuel P. Midkiff

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

This page intentionally left blank.

A Case Study of Fortran in Computational Science and Engineering

José Moreira Sam Midkiff
{moreira,midkiff}@watson.ibm.com

IBM T. J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Abstract

Fortran has been the dominant language in scientific and engineering computing for nearly 40 years. Fortran's latest widely-available form, Fortran 90, provides important facilities for developing large CSE applications: array language, abstract data types, modularity, and encapsulation. Object-oriented programming has been successfully accomplished in Fortran 90. All this has been achieved while maintaining a major design goal throughout the evolution of the language: high performance on numerical-intensive computing. In this paper, we present a case study of Fortran 90 in a CSE application. We show that its features facilitate the development of applications. We also show that the right compiler can generate very efficient code from Fortran 90 source, both in serial and parallel forms. Finally, we discuss some of the issues involved in choosing the right language for an application, and the topic of language interoperability.

1 Introduction

Fortran, originally an acronym for the *IBM Mathematical FORmula TRANslating System*, was first developed by a team led by John Backus of IBM, during the mid-fifties. The first implementation became available for the IBM 704 computer in 1957. From the beginning, the language targeted numerical computation, and this focus has not changed in 40 years of evolution. By some measures, more than 90% of current scientific computation uses Fortran [5].

In 1966, the version known as Fortran 66 became the first language ever to be standardized. A major revision was approved in 1978, in the form of Fortran 77. The latest standard, Fortran 95 [12] represents only a minor revision over the widely available Fortran 90 [1], which substantially extended and modernized the language in 1991. Work is well under way for the next standard, Fortran 2000, to be published in the year 2001.

This paper is basically a case study of the applicability of the two main new features of Fortran 90, the array language and abstract data types, to computational science and engineering (CSE). We collectively call scientific and engineering applications, *technical applications*. Technical applications are typically intensive in numerical computing, often manipulating large collections of data in the form of arrays. Technical programmers have, for a long time, cherished the ability of Fortran implementations to produce efficient code for numerical computations. In this paper, we show that the new features of Fortran 90 can be used without compromising efficiency, provided that the appropriate compiler technology is used. Furthermore, we show that compilers can successfully parallelize Fortran 90 code, making it run even faster when executing on a multiprocessor system.

Although object-orientedness was not directly built into Fortran 90, its data abstraction and modularity features allow it to simulate many of the features of C++. This has brought object-oriented programming to the realm of numerical computing in Fortran 90 [6]. Conversely, much progress has been made towards efficient numerical computing in C++ [9, 11], which has sparked a debate on which language is more appropriate for developing large technical applications [2]. From a performance perspective, the main difference between the two languages is that the features of Fortran 90 for supporting technical computing (*e.g.*, the array language and the `COMPLEX` data type) are integral part of the language and have their semantics exposed to the compiler. In C++, such features are provided in the form of class libraries, which are processed by the compiler just like any other classes. We claim, and in one case demonstrate through experimentation, that this additional knowledge available to the compiler gives Fortran 90 a performance advantage.

It is very likely that the final choice of which language to use for developing a technical application will be based in what we call *non-technical issues*: availability of programmers, features of development environments, etc. Moreover, in large applications it is common to use multiple programming languages, using each one to perform the tasks it does better.

The rest of this paper is organized as follows. Section 2 describes the CSE application we use to experiment with Fortran 90 and C++. Section 3 describes the Fortran 90 implementation of this application, showing that this language remains faithful to its FORmula TRANslating origins. In Section 4 we present and analyze the results from our performance evaluation of Fortran 90 and C++ versions of our application. Finally, in Section 5 we present some discussion and concluding remarks.

2 A CSE Application

As a working example, we consider a CSE application from the field of electromagnetics. In a shielded microstrip structure, shown in Figure 1, n parallel microstrips run inside a homogeneous dielectric, with permittivity ϵ_d [3]. A conducting shield completely encloses the microstrips and the dielectric. The shield is connected to a reference potential ($V_{\text{shield}} = 0$). Each microstrip k is at a voltage V_k with respect to the shield. We want to compute the value of the potential field $\Phi(x, y)$ in each point of the structure.

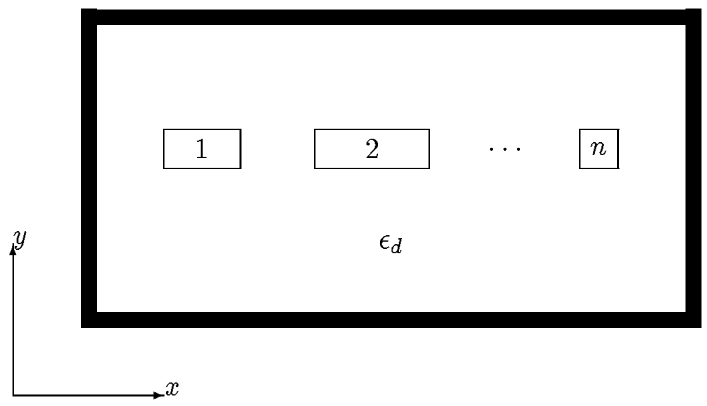


Figure 1: A shielded microstrip structure.

Because the dielectric is free of charge, Maxwell's equation for the potential in the dielectric is reduced to Laplace's equation:

$$\nabla^2\Phi(x,y) \equiv \frac{\partial^2\Phi(x,y)}{\partial x^2} + \frac{\partial^2\Phi(x,y)}{\partial y^2} = 0. \tag{1}$$

The potential field $\Phi(x,y)$ is subject to boundary conditions $\Phi = 0$ along the shield and $\Phi = V_k$ along microstrip k.

Equation 1 is expressed in a continuous domain, that covers the extent of the microstrip structure. To solve this equation numerically, we discretize the domain into a rectangular *grid* (also called a *mesh*) of evenly spaced points. This discretization is illustrated in Figure 2. The grid has $w+1$ points along axis x and $h+1$ points along axis y. The step s of the grid is the distance between two neighboring points along the x or y axes. We are concerned with finding the value of Φ in points (x_i, y_j) for $i = 0,\ldots,w$ and $j = 0,\ldots,h$. Note that the edges of the shield and microstrips must be aligned with the grid. We can represent a microstrip k by the set of grid points T_k that it spans. For a microstrip with rectangular cross-section, it suffices to know the locations of its lower-left and upper-right corners.

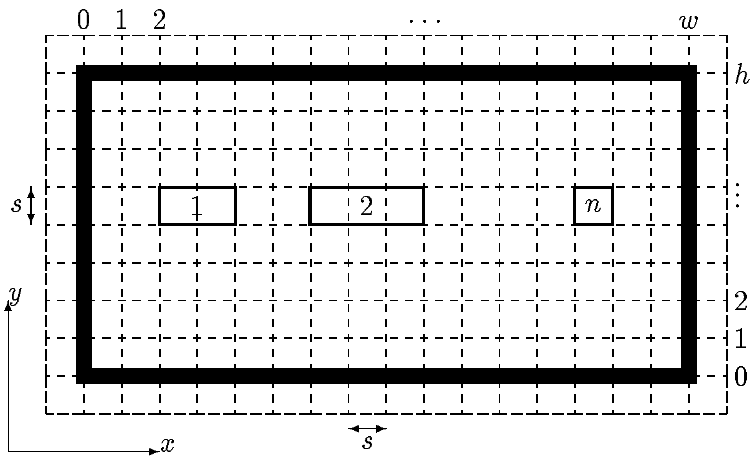


Figure 2: A shielded microstrip structure.

With the grid in place we can now approximate the second order differential operators $\partial^2/\partial x^2$ and $\partial^2/\partial y^2$ in Equation (1) by *difference* operators [8]. Let $\Phi_{i,j}$ denote the value of Φ in point (x_i, y_j) of the grid. The we can write

$$\frac{\partial^2\Phi_{i,j}}{\partial x^2} \simeq \frac{\Phi_{i+1,j} - 2\Phi_{i,j} + \Phi_{i-1,j}}{s^2} \tag{2}$$

$$\frac{\partial^2\Phi_{i,j}}{\partial y^2} \simeq \frac{\Phi_{i,j+1} - 2\Phi_{i,j} + \Phi_{i,j-1}}{s^2} \tag{3}$$

By substituting Equations (2) and (3) into Equation (1), we get

$$\Phi_{i+1,j} + \Phi_{i-1,j} + \Phi_{i,j+1} + \Phi_{i,j-1} - 4\Phi_{i,j} = 0 \tag{4}$$

in every point (i,j) of the dielectric. The boundary conditions say that $\Phi_{i,j} = 0$ on those points (i,j) along the edges of the shield, and that $\Phi_{i,j} = V_k$ on those points (i,j) belonging to microstrip k. One approach to solving Equation (4) is through an iterative method called *Jacobi relaxation*: We rewrite Equation (4) as

$$\Phi_{i,j} = \frac{1}{4}(\Phi_{i+1,j} + \Phi_{i-1,j} + \Phi_{i,j+1} + \Phi_{i,j-1}) \tag{5}$$

We then start with some arbitrary values of $\Phi_{i,j}$ (*e.g.*, zeroes) in the dielectric and the appropriate values in the boundary points. Then, at each step of the relaxation we recompute the values of $\Phi_{i,j}$, for all points (i,j) in the dielectric, using Equation (5). The iterations are repeated until a solution is converged to. See [8] for a discussion of Jacobi relaxation and other related methods. The pseudo-code for this algorithm is shown in Figure 3.

procedure *Jacobi*$(\Phi, \Phi', w, h, V, T, n, \varepsilon)$ {

S_1 $\quad r \leftarrow 2\varepsilon$

$\quad$ **while** $(r > \varepsilon)$ {

S_2 $\quad\quad \Phi'_{i,j} \leftarrow \frac{1}{4}(\Phi_{i+1,j} + \Phi_{i-1,j} + \Phi_{i,j+1} + \Phi_{i,j-1}),\ \forall i = 1,\ldots,w-1,\ \forall j = 1,\ldots,h-1$

S_3 $\quad\quad (\Phi'_{i,j} \leftarrow V_k,\ \forall(i,j) \in T_k),\ \forall k = 1,\ldots,n$

S_4 $\quad\quad \Phi_{i,j} \leftarrow \frac{1}{4}(\Phi'_{i+1,j} + \Phi'_{i-1,j} + \Phi'_{i,j+1} + \Phi'_{i,j-1}),\ \forall i = 1,\ldots,w-1,\ \forall j = 1,\ldots,h-1$

S_5 $\quad\quad (\Phi_{i,j} \leftarrow V_k,\ \forall(i,j) \in T_k),\ \forall k = 1,\ldots,n$

S_6 $\quad\quad r \leftarrow \sum_{i,j} |\Phi_{i,j} - \Phi'_{i,j}|/((w+1) \times (h+1))$

$\quad$ }

}

Figure 3: Algorithm for computing $\Phi_{i,j}$ through Jacobi relaxation.

We first initialize, in statement S_1, the current error r with twice the desired error ε. The iterative solver will stop when the current error is less than or equal to the desired error. In statement S_2 we perform the relaxation proper, by computing new values from $\Phi_{i,j}$ into $\Phi'_{i,j}$. Note that we compute new values for all points in the grid internal to the shield and then we reset the boundary conditions for the microstrips in S_3. We repeat the relaxation in statement S_4, now computing the new values into Φ. We again reset the boundary conditions for the microstrips in S_5. By performing two relaxations in each iteration of the **while** loop, we avoid having to copy values from Φ' back to Φ. Finally, we evaluate the current error in statement S_6 by comparing two consecutive solutions, Φ and Φ'. We first sum the absolute value of the difference between the two solutions over all grid points. We then compute the error by dividing this sum by the number of grid points, $(w+1) \times (h+1)$.

We consider two modes of operation in our shielded microstrip structure: DC and AC. In the DC mode, the voltage at microstrip k is a constant value V_k. In this case, the potential field $\Phi(x,y)$ is a real function of x and y, and all operations in **procedure** *Jacobi* can be performed with real arithmetic. In the AC mode, the voltage at microstrip k is a function of time of the form $V_k \cos(\omega t - \phi)$. This voltage can be represented by a complex number $V_k \cos(\phi) + iV_k \sin(\phi)$. In this case, the field $\Phi(x,y)$ is a complex function of x and y, and all operations in **procedure** *Jacobi* must be performed with complex arithmetic.

3 Language considerations

High performance for numerical applications has always been a major design goal for Fortran. Up to and including Fortran 77, other features such as usability were given only limited attention. In the forty-plus years of Fortran's existence, the size of applications in general, and of technical applications in particular, has grown substantially. Large Fortran systems of a million or more lines of code exist. As a consequence, the design goals of the language broadened with the Fortran 90 specification. In particular, the concepts of encapsulation, modularity, and abstract data types, features proven useful in other languages for managing the complexity of large systems, have been introduced to Fortran. We illustrate several of the new features of Fortran 90 using a program that implements the previously discussed microstrip application.

Perhaps the most widely used of Fortran's new features is the array language. Fortran 90 allows an entire array to be manipulated as a single object. All arithmetic operations, and most of the Fortran intrinsic functions, are extended to handle arrays. Moreover, the shape (extents) of an array can be defined at run-time. As an example, consider the following code fragment, from the main program of our microstrip application:

```
INTEGER     w, h
COMPLEX*16  Phi, PhiPrime
ALLOCATABLE Phi(:,:), PhiPrime(:,:)

READ *, w, h
ALLOCATE(Phi(0:w,0:h),PhiPrime(0:w,0:h))
Phi = 0.0
CALL UpdateStrips(Phi,w,h,V,T,n)
PhiPrime = Phi
```

`Phi` and `PhiPrime` are declared as two-dimensional `ALLOCATABLE` arrays of `COMPLEX*16` elements. During execution, the values of `w` and `h` (the grid extents) are read. After that, the arrays `Phi` and `PhiPrime` can be created with the appropriate shape, using the `ALLOCATE` statement. The statement `Phi = 0.0` initializes all elements of `Phi` to zero. After the potential on the microstrips is initialized by the `UpdateStrips` subroutine, the statement `PhiPrime = Phi` assigns the value of each element of `Phi` to the corresponding element of `PhiPrime`.

Statement S_6 of **procedure** *Jacobi* can be implemented with a single Fortran 90 statement:

```
error = SUM(ABS(Phi - PhiPrime))/((w+1)*(h+1))
```

This statement performs an element by element subtraction between `Phi` and `PhiPrime`, which generates another two-dimensional array as a result. The intrinsic function `ABS` computes the absolute value of all elements of the `Phi - PhiPrime` array (which logically results in yet another two-dimensional array), while the intrinsic `SUM` computes the sum of all these absolute values.

Sections of arrays can be directly represented in the language, and used in the same way as a defined array of the same shape. Thus, the expression `Phi`$(l_1:u_1:s_1, l_2:u_2:s_2)$ defines a two-dimensional array consisting of rows l_1 through u_1 in steps of s_1, and columns l_2 through u_2 in steps of s_2, of array `Phi`. In many situations, the step is 1 and can simply be omitted. Using array sections, we can write statement S_2 of **procedure** *Jacobi* as a single Fortran 90 statement:

```
PhiPrime(1:w-1,1:h-1) = 0.25*(Phi(2:w,1:h-1) + Phi(0:w-2,1:h-1) +
                              Phi(1:w-1,2:h) + Phi(1:w-1,0:h-2))
```

The Fortran 90 array language allows common array operations to be expressed succinctly and naturally in the language.

Much of the convenience of the Fortran 90 array language can be duplicated in C++ by defining an array class, and overloading the standard arithmetic operators so that they are defined with array operands. Some success with this approach has been reported in [9]. The advantage Fortran has is that the meaning of arrays and of the overloaded operators are known to the language processor (compiler), whereas in C++ the class is simply another collection of data and routines. The additional knowledge of the Fortran compiler can be exploited to generate more highly optimized code. The overall performance advantages of Fortran are quantified in Section 4.

Fortran 90 has adopted the concepts of derived data types and encapsulation. Like arrays, derived types allow data to be defined in ways that reflect the properties of the underlying system being modeled. Encapsulation is the ability to group data and operations on that data into a logical unit, or *module*, and only expose the external interface components of the data and operations. Encapsulation is supported in Fortran 90 by the `MODULE` construct, which provides some, but not all, of the functionality of a C++ class. The derived data types and the `MODULE` construct combine to provide a mechanism to build abstract data types. We illustrate these concepts with the `MODULE Microstrip`, developed for our application and shown in Figure 4.

First, derived type `Point` is declared with the `TYPE` construct, to represent a point in a grid. `X` and `Y` are the coordinates of the point. Then, a second derived type `Strip` is declared to represent a microstrip. A `Strip` is defined by two points, `LL` and `UR`, which represent the positions of the lower-left and upper-right corners of the microstrip, respectively. These two data types are encapsulated in `MODULE Microstrip`, together with two functions, `UpdateStripsComplex` and `UpdateStripsReal`, that operate on microstrips. These functions implement statements S_3 and S_5 of **procedure** *Jacobi*, one for the AC mode of operation and the other for the DC mode. The loop

```
DO i = 1, n
   Phi(T(i)%LL%X:T(i)%UR%X,T(i)%LL%Y:T(i)%UR%Y) = V(i)
END DO
```

sets the grid points of each microstrip `T(i)` to voltage `V(i)`. Note that `T(i)%LL%X` denotes the `X` coordinate of the lower-left corner of microstrip `T(i)`.

Another very useful feature of `MODULE` is the ability to overload `SUBROUTINE` and `FUNCTION` names. This is done via the `INTERFACE` and `MODULE PROCEDURE` statements. Thus, in `Microstrip` we associate both `UpdateStripsReal` and `UpdateStripsComplex` with the name `UpdateStrips`. The Fortran compiler, upon encountering a call to `UpdateStrips`, will examine the arguments, and call the `MODULE PROCEDURE` which takes the same arguments.

Not shown in this example are the use of `PUBLIC` and `PRIVATE` specifiers to allow the `MODULE` programmer to describe the scope of objects defined in the `MODULE`. This is an important feature of `MODULE`s, as it separates the interface component from the implementation component. Any program unit outside a `MODULE` that wishes to access the (`PUBLIC`) data and routines of the `MODULE` should include the `USE` statement (*e.g.*, `USE Microstrip`). The complete code for the implementation of **procedure** *Jacobi*, illustrating the concepts introduced so far, is shown in Figure 5.

Fortran 90 has several other features that contribute to the goal of efficient program execution. The first of these is the concept of an *intrinsic* routine. Intrinsics are standard functions whose meanings are known to the compiler. Intrinsics allows the compiler to know what data is changed by the function, and how that data is changed. This in turn allows the compiler to do optimization across an intrinsic call, and to recreate inline code that is equivalent in meaning to the intrinsic. The inlining is very important, as it allows the compiler to optimize the calling code and the intrinsic together. Inlining also allows the overhead of a function call/return to be eliminated. Of special importance to parallelizing compilers, an intrinsic function can be specialized for a particular execution environment.

Another optimizing feature is a rich collection of base data types. The array operations discussed earlier are one example of this. Of particular interest to technical computing in general, and to our application in particular, is the `COMPLEX` data type, and all the intrinsics and standard numeric operations that support this data type. In C++, support for complex numbers is provided in a class library. While this approach delivers very good notation convenience, Fortran once again has the advantage that the `COMPLEX` data type and its semantics are known to the compiler. In Section 4 we discuss the performance implications of this difference.

The use of `INTERFACE` blocks allows a compiler to have some insight into the behavior of a procedure whose code is either not available (*e.g.*, a library function for which only the binary is available) or is available only through inter-procedural analysis. For example, in the main program of our application the interface:

```
MODULE Microstrip
IMPLICIT NONE

TYPE Point
   INTEGER        X
   INTEGER        Y
END TYPE Point

TYPE Strip
   TYPE(Point)    LL
   TYPE(Point)    UR
END TYPE Strip

INTERFACE UpdateStrips
   MODULE PROCEDURE UpdateStripsComplex
   MODULE PROCEDURE UpdateStripsReal
END INTERFACE

CONTAINS

SUBROUTINE UpdateStripsComplex(Phi,w,h,V,T,n)
INTEGER           w, h, n
COMPLEX*16        Phi(0:w,0:h), V(1:n)
TYPE(Strip)       T(1:n)

INTEGER   i,
DO i = 1, n
   Phi(T(i)%LL%X:T(i)%UR%X,T(i)%LL%Y:T(i)%UR%Y) = V(i)
END DO

END SUBROUTINE UpdateStripsComplex

SUBROUTINE UpdateStripsReal(Phi,w,h,V,T,n)
INTEGER           w, h, n
REAL*8            Phi(0:w,0:h), V(1:n)
TYPE(Strip)       T(1:n)

INTEGER   i,
DO i = 1, n
   Phi(T(i)%LL%X:T(i)%UR%X,T(i)%LL%Y:T(i)%UR%Y) = V(i)
END DO

END SUBROUTINE UpdateStripsReal

END MODULE Microstrip
```

Figure 4: An example of a Fortran 90 Module.

```
SUBROUTINE Jacobi(Phi,PhiPrime,w,h,V,T,n,eps)
USE Microstrip
INTEGER         w, h, n
COMPLEX*16      Phi(0:w,0:h), PhiPrime(0:w,0:h), V(1:n)
REAL*8          eps
TYPE(Strip)     T(1:n)

REAL*8          error
error = 2*eps

DO WHILE (error > eps)
   PhiPrime(1:w-1,1:h-1) = 0.25*(Phi(2:w,1:h-1) + Phi(0:w-2,1:h-1) +
                                 Phi(1:w-1,2:h) + Phi(1:w-1,0:h-2))
   CALL UpdateStrips(PhiPrime,w,h,V,T,n)
   Phi(1:w-1,1:h-1) = 0.25*(PhiPrime(2:w,1:h-1) + PhiPrime(0:w-2,1:h-1) +
                            PhiPrime(1:w-1,2:h) + PhiPrime(1:w-1,0:h-2))
   CALL UpdateStrips(Phi,w,h,V,T,n)
   error = SUM(ABS(Phi - PhiPrime))/((w+1)*(h+1))
END DO

END SUBROUTINE Jacobi
```

Figure 5: Implementation of **procedure** *Jacobi* in Fortran 90.

```
INTERFACE
   SUBROUTINE Jacobi(Phi,PhiPrime,w,h,V,T,n,eps)
   USE Microstrip
   INTENT(IN), INTEGER :: w, h, n
   INTENT(INOUT), COMPLEX*16 :: Phi(0:w,0:h), PhiPrime(0:w,0:h)
   INTENT(IN), COMPLEX*16 :: V(1:n)
   INTENT(IN), REAL*8 :: eps
   INTENT(IN), TYPE(Strip) :: T(1:n)
   END SUBROUTINE Jacobi
END INTERFACE
```

allows the compiler to know that `Jacobi` is a Fortran 90 routine, to check that the arguments passed to `Jacobi` are of the correct type, and to know that only `Phi` and `PhiPrime` are changed. The writer of a library can place this information in a `MODULE`, and thereby expose a great deal of information useful for error checking and optimization.

C++ allows much of the functionality we have described, and significantly extends it in some areas (*e.g.*, with the *template* facility). Arguably, the C++ functionality places more generality in the hands of the application programmer. The Fortran implementation of the features, however, has significant advantages in the domain of numerical coding if performance is an important consideration. First, Fortran argument semantics specify that arguments are not aliased. This, in conjunction with Fortran's more restrained pointer semantics and the use of `INTERFACE` allows more data-flow information to be exposed to the compiler. It may be impossible or very difficult for a C++ compiler to gather the same information. Second, the richer set of base data types, and the use use of intrinsics allows the compiler to infer more about the meaning of a program than can be done, in general, by a C++ compiler. Finally, a call to a Fortran procedure can always be resolved at compile time and implemented directly. When programming with inheritance, a call to a C++ method must often involve a lookup in a table of pointers to procedures (*virtual function table* – VFT). This indirection is expensive at run-time, and by obscuring the actual calling hierarchy of the program reduces the ability of the compiler to globally analyze the program. The use of VFTs is part of the cost of C++'s superior extensibility.

One final issue to be discussed is the ease with which parallel programs can be written. While we will not argue that parallel programming is a trivial task, we will argue that Fortran provides superior support. There are at least three industrially supported efforts to enable the writing of parallel Fortran codes – MPI [10], HPF [4] and OpenMP [7]. We chose to parallelize our application with HPF, as it was the simplest approach in our experimental environment.

With the addition of eight HPF `DISTRIBUTE` directives we were able to parallelize the Microstrip application. All these directives are of the form

```
!HPF$ DISTRIBUTE (*,BLOCK) ::  Phi, PhiPrime
```

and they specify a `(*,BLOCK)` distribution for the `Phi` and `PhiPrime` arrays wherever they appear in the code. The "`*`" distribution for the columns allowed the column major accesses to be maximized within a given processor, thereby enhancing cache locality. The "`BLOCK`" distribution along the rows enabled parallelism. This distribution, when applied to four processors, is illustrated in Figure 6. The `Phi` and `PhiPrime` arrays were chosen for distribution because they are the operands in the main computations of the application.

```
!HPF$ DISTRIBUTE (*,BLOCK) ::  Phi, PhiPrime
```

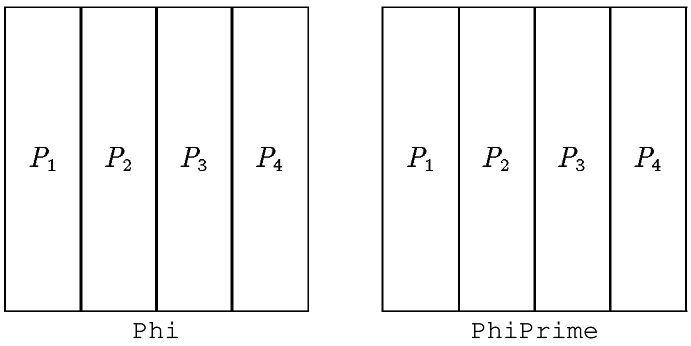


Figure 6: Example of an HPF distribution on four processors (P_1, P_2, P_3, and P_4).

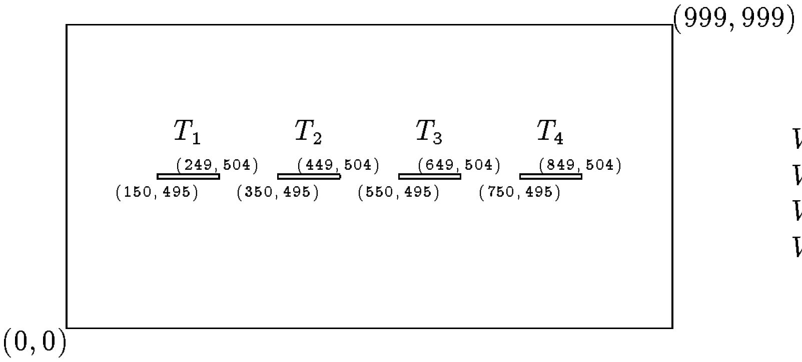
Although it would be incorrect to say that this example is typical of the effort to parallelize a standard Fortran code, it is indicative of the fact that the same sort of attention paid to performance and optimization in Fortran is beneficial to the task of parallelization.

4 Performance issues

In this section we analyze the performance of our Fortran implementation of the microstrip application. We also compare it to the performance of a C++ version of the same application. We execute each version in both an AC mode of operation and a DC mode of operation. We show benefits of having features like the `COMPLEX` data type built directly into the language. We also demonstrate the importance of compiler technology, in particular dependence analysis, in achieving high performance with array operations. Finally, we show that the Fortran code can be efficiently parallelized with current compiler technology.

4.1 Input problem

We used our microstrip application to analyze the structure shown in Figure 7. The structure is laid on a 1000×1000 grid ($w = h = 999$), and it has four microstrips. Each microstrip is 10 grid-steps high and a 100 grid-steps wide. The grid positions of the lower-left and upper-right corner of each microstrip are indicated in the figure. For the AC mode of operation, the voltage on microstrip k is $\cos(\omega t - (k-1)90^\circ)$ volts. For the DC mode of operation, the voltage on all microstrips is 1 volt.



AC	DC
$V_1 = \cos(\omega t)$	$V_1 = 1$
$V_2 = \sin(\omega t)$	$V_2 = 1$
$V_3 = -\cos(\omega t)$	$V_3 = 1$
$V_4 = -\sin(\omega t)$	$V_4 = 1$

Figure 7: Input problem.

4.2 Experimental environment

We measured the performance of our application on two different partitions of an IBM RS/6000 SP. One partition consists of 16 thin nodes, while the other partition consists of 4 wide nodes. Both thin and wide nodes have a 67 MHz POWER2 processor. The wide nodes have more cache (256 kB *vs.* 64 kB) and better memory bandwidth (600 MB/s *vs.* 150 MB/s) than the thin nodes.

On each partition, and for each mode of operation (AC or DC) we ran four different versions of our benchmark: (i) a serial Fortran version compiled with dependence analysis, (ii) a serial Fortran version compiled without dependence analysis, (iii) a serial C++ version, (iv) a parallel Fortran version, compiled with the HPF compiler. Obviously, only the parallel Fortran version could run on more than one node. Each run consisted of several iterations of our *Jacobi* solver, and we report our results in terms of time per iteration. The baseline for our comparisons is the fastest serial version, which was generated by the Fortran compiler with dependence analysis. The time for the HPF version on 1 node is nearly identical to this best serial version. If t_b is the timer per iteration of the baseline version, then we compute the speedup of a version with time per iteration t_v through the expression $S = t_b/t_v$.

4.3 Results

Figure 8 shows the results for the 4 combinations of wide and thin nodes, and AC and DC mode of operation. Each plot shows the speedup of each version (serial Fortran without dependence analysis – identified as F90, C++, and HPF) with respect to the baseline (Fortran with dependence analysis). For HPF, we report the performance for different numbers of processors. The height of each bar is proportional to the speedup for that version, while the number on the top of the bar is the execution time, in seconds, for each iteration of the *Jacobi* solver.

We observe a very good scalability of the HPF version of our application. When running on 16 nodes, this version achieves a speedup of 11 for the AC mode and of 9 for the DC mode. The speedup for the AC mode is better because the operations with `COMPLEX` data types result in a larger granularity for each grid

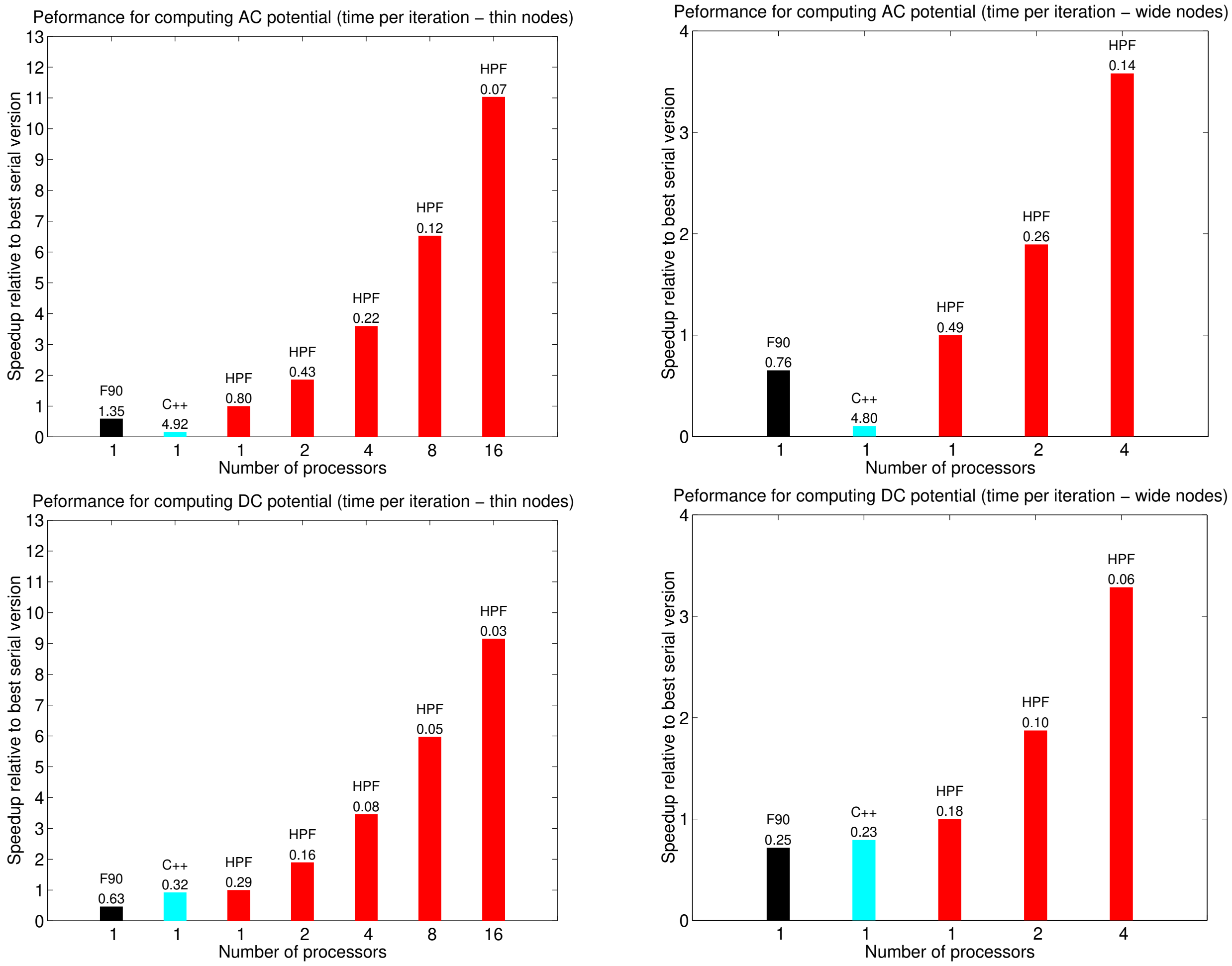


Figure 8: Performance results.

point. The POOMA project [9] has successfully implemented a framework for automatic parallelization of scientific C++ applications. When running an application similar to ours (in DC mode) in the same environment, they report a speedup of 6.5 on 16 nodes. The single-node Mflops rate is about 75% of ours.

Our C++ version implements the array operations explicitly, through doubly-nested loops. The use of expression templates [11] and chained-expression objects [9] has resulted in array classes that approach the efficiency of explicit loops. The C++ version is competitive with the best serial Fortran version for the DC mode of operation. It achieves 90% of the baseline performance on a thin node and 80% on a wide node. However, in the AC mode of operation, the C++ version is 6 and 10 times slower than the baseline, for the thin and wide node respectively. In C++, complex arithmetic is implemented in a class library, with no special treatment by the compiler. In Fortran, complex arithmetic is integral to the language. By understanding the semantics of complex arithmetic, the compiler can generate more efficient code.

Finally, we note that dependence analysis is important to extract good performance from code with array operations. The Fortran version compiled without dependence analysis is 1.5 to 2 times slower than the baseline. In fact, for the DC mode, Fortran without dependence analysis is slower than C++. Without dependence analysis, an expression of the form:

```
C(1:n) = C(1:n) + A(1:n) * B(1:n)
```

is first translated in a sequence of more elementary operations:

S_1 `T1(1:n) = A(1:n) * B(1:n)`
S_2 `T2(1:n) = C(1:n) + T1(1:n)`
S_3 `C(1:n) = T2(1:n)`

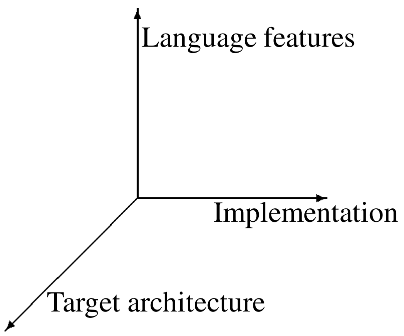
and then each of these operations is implemented with a `do`-loop. This involves unnecessary manipulation of temporary vectors `T1` and `T2`. Simple name analysis shows that there is just a flow-dependence from S_1 to S_2, and therefore those two statements can be combined. However, a more detailed *range* analysis (which looks at the range of indices inside subscript) has to be used to show that there are no anti- or output-dependences from S_2 to S_3. With dependence analysis, the operation can be translated directly to a single `do`-loop:

```
do i = 1, n
  C(i) = C(i) + A(i)*B(i)
end do
```

In the above example, this optimization reduces the number of reads from `5n` to `3n`, and the number of writes from `3n` to `n`.

5 Discussion

The choice of a language for implementing an application has both technical and non-technical components. Non-technical components include the availability of programmers familiar with a language, the set of languages already supported in a site, and the quality of accessible development environments. These issues, although important, are outside the scope of this paper. Technical components in the choice of a programming language can be viewed as axes in a three-dimensional space, as shown in Figure 9.



- **Language features:** what the language provides to simplify application development.
- **Implementation:** how good the available implementations of the language are.
- **Target architecture:** the effects of the target machine in intensifying or minimizing the benefits of the language.

Figure 9: The three axes of choosing a programming language.

The first axis represents the features in the language itself, and their suitability for the project at hand. It is clear that domain specific languages (like Fortran, Cobol, RPG and SQL) are not particularly well suited for projects outside of their domain. Less obvious are the tradeoffs involved in using a general purpose language, like C++ or Java$^{(TM)}$, to implement applications in the domain of a domain specific language. Many modern programming language features, like encapsulation, modularity, and abstraction, have been included in Fortran 90. Although the language remains technical-oriented, it can now tackle large problems in a more structured way. On the other hand, the POOMA project [9] has been successful in achieving good results in the implementation of technical application in C++. Arguments have been made both ways for the benefits of using either Fortran 90 or C++ for large technical applications [2, 6]. It is our position, supported by our experimental results, that the facilities built into the language still give Fortran 90 a performance advantage.

The second axis in the graph is the quality of implementation of the language. This is particularly important in languages like Fortran, where performance is a major, if not the major, reason for its use. Early implementations of Fortran 90 did not use dependence analysis to determine if the right hand side in the statement `a(1:n-1) = a(2:n)` could be immediately copied into the left hand side, or if it was first necessary to copy `a(2:n)` into a temporary, and then copy the temporary into `a(1:n-1)`. Making the wrong choice and doing the copy to a temporary will double the number of necessary memory moves. Similarly, in many cases where a copy appears necessary, reversing the order that the implicit loop executes will make a direct transfer legal. The dramatic effects this lack of analysis and optimization has on program execution efficiency was shown in Section 4, when dependence analysis was turned off. The conclusion to be drawn from this is that a relatively poor implementation of a domain specific language may perform worse than a good implementation of a more general language. Thus the quality of implementation of available languages must be considered when making the choice.

The third axis in the graph is the target architecture. The function of a high-level language compiler is to map the high-level language onto hardware. Hardware that is more capable (*i.e.*, has more memory bandwidth, a larger cache, more functional units) presents more resources for a smart compiler to exploit. In Section 4 we saw the C++ performance go from 80% to 90% of the best Fortran performance, when we moved from wide to thin SP nodes. Our experience shows that the less capable the hardware, the smaller the performance differences among programming languages.

Finally, we discuss a common trend in the development of large applications: the use of multiple programming languages. Since a large application is bound to span several domain, it is natural to use multiple programming languages, choosing the one more appropriate for each part of the application. For example, C++ can be used for high-level coordination, file input/output, and data-base access. Fortran can be used for the numerical-intensive kernels, that perform most of the computation and could be parallelized. Java$^{(TM)}$ can handle communication and data-transfer with other machines. For this menagerie to work together, good language interoperability is a key factor. It must be possible to easily pass data and control from a routine written in one language to a routine written in another. Java$^{(TM)}$, for example, defines a native interface for interoperability with C and C++. In general, it is easier for languages to work together when all the compilers are from the same family. For example, the IBM XL family of compilers for AIX supports interoperability among C, C++, Fortran, and Pascal routines.

References

[1] 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.

[2] J. R. Cary, S. G. Shasharina, J. C. Cummings, J. V. W. Reynders, and P. J. Hinker. Comparison of C++ and Fortran 90 for object-oriented scientific programming. *Computer Physics Communications (Netherlands)*, 105(1):20–36, September 1997.

[3] T. C. Edwards. *Foundations for Microstrip Circuit Design*. John Wiley & Sons, Chichester, NY, 1992.

[4] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[5] C. Lazou and P. Anderton. Fortran: past, present and future developments. *Scientific Computing World*, (17):31–32, April 1996.

[6] C. D. Norton, V. K. Decyk, and B. K. Szymanski. High performance object-oriented scientific programming in Fortran 90. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN*, March 14–17 1997.

[7] OpenMP Organization. OpenMP: A proposed standard API for shared memory programming. White paper at `http://www.openmp.org/`.

[8] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in FORTRAN: The Art of Scientific Computing*. Cambridge University Press, 1992.

[9] J. V. W. Reynders, J. C. Cummings, M. Tholburn, P. J. Hinker, S. R. Atlas, S. Banerjee, M. Srikant, W. F. Humphrey, S. R. Karmesin, and K. Keahey. POOMA: A framework for scientific simulation on parallel architectures. In *Proceedings of First International Workshop on High Level Programming Models and Supportive Environments, Honolulu, HI*, pages 41–49, April 16 1996. Technical report available at `http://www.acl.lanl.gov/PoomaFramework/papers/papers.html`.

[10] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, Cambridge, MA, 1996.

[11] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[12] X3J3. *International Standard Programming Language Fortran, ISO/IEC 1539-1:1996*. Document X3J3/96-007R1 available at `http://www.uni-karlsruhe.de/~Fortran95/96-007r1/`.