

Project Fortress

A multicore language for multicore processors

By Eric Allen, David Chase, Christine Flood, Victor Luchangco,
Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr.

The computing world is undergoing a sea change. As computers become more powerful, high performance computing, an endeavor historically relegated to national labs and government institutions, is becoming mainstream. For example, on a daily basis, search engine companies around the world employ farms of machines executing parallel code. Additionally, global networks of privately-owned computers — think *Folding@home* and *SETI@home* — tackle some of the most computationally challenging problems of modern science. At the same time, microchip manufacturers, in a continuing effort to improve performance, are designing new chips that contain increasing numbers of cores (processors) on a single chip. Yet ironically and unfortunately, modern programming languages are ill-equipped to keep pace with these changes. To exploit multiple cores, programs must execute not only in parallel, but with a degree of parallelism that varies from platform to platform. Portable programs cannot be tied to a specific number of cores, because that number will vary across machines.

The same issue arises when writing programs for large clusters: programs must execute in parallel on a large and varying number of processors. Most languages don't directly support any notion of parallelism, requiring the programmer to graft additional notation such as *MPI* or *OpenMP* onto programs. Even languages with support for parallelism, such as *Java*, support “heavyweight” notions of parallelism, where threads are explicitly defined and allocated, and the cost of creating a new thread is typically high.

Programs must execute not only in parallel, but with a degree of parallelism that varies from platform to platform. Programmers need help.

Moreover, to write effective parallel programs, the programmer needs help. It's notoriously easy to inadvertently create race conditions when writing parallel programs. Existing language features designed to control parallelism, such as locks,

are inadequate because they combine poorly. For example, if an application uses two libraries, where each maintains a locking discipline over its respective data structures, there's no way to ensure that the composition of the libraries in an application remains free of race conditions or deadlocks, since the libraries do not use the same locks.

Compiler writers have tried to find ways to execute programs in parallel automatically, without altering the semantics of the underlying language. Unfortunately, this kind of “auto-parallelization” is difficult to do because the language semantics are often violated by parallelization in minor ways.

To address these shortcomings of modern programming languages, Sun Labs is designing *Fortress*, a new programming language for scientific computing. *Fortress* was initially developed as part of the *DARPA HPCS* project to achieve new levels of productivity and performance in high performance computing. It is now an open source project with a prototype implementation freely available at <http://fortress.sunsource.net>. An active community of programmers external to Sun contributes code and provides suggestions for refining the language semantics.

Fortress includes built-in support to easily define and control parallelism in a portable manner, and also includes many other language features that make it well-suited for both scientific and general-purpose programming. For the scientific programmer, Fortress provides a syntax based on standard mathematical notation, and includes support for concepts that are important for scientific computation, such as vectors, matrices, and physical units (meters, seconds, and so on). For general-purpose programming, Fortress includes a trait-based type system that supports multiple implementation inheritance, first-class overloaded functions with multiple dynamic dispatch, and various facilities for “programming in the large,” such as contracts, automated unit testing, and a component system for developing, deploying, and upgrading libraries and other programs.

Let’s explore the features of Fortress, as well as the principles underlying the design of those features, and how you can participate in its continued development.

High-level High-performance Computing

Unlike most parallel languages that augment a sequential language with parallel features or annotations, Fortress is implicitly parallel wherever possible, and provides constructs and annotations to serialize execution when necessary. As a result, a compiler or virtual execution environment need not concern itself with determining whether executing a program in parallel is *permissible*, only whether doing so is *advantageous*. (To be sure, this latter question is not trivial; parallel execution might not improve performance if all available processors are already fully-utilized, for example, or if moving data to an underutilized processor is more costly than doing the computation.)

TO DO FOR FORTRAN WHAT JAVA DID FOR C

The name “Fortress” is derived from the intent to produce a “secure *Fortran*,” a language for scientific computation that provides built-in support for parallelism, along with abstraction and type safety in line with modern programming language principles, without compromising performance.

Fortress is intended to enable scientific programmers to write safer, more efficient and more portable code by providing many of the benefits that the *Java* programming language brought to C programmers: a rich object-oriented type system with static type checking, a virtual execution environment with automatic memory management and dynamic compilation, extensive libraries, and built-in support for parallel programming.

Indeed, Fortress goes significantly beyond *Java* on each of these aspects. However, Fortress is not derived from *Fortran*, nor from the *Java* programming language. Rather, it is an entirely new language designed to meet the needs of scientific programmers.

TABLE ONE: Common Generators in Fortress

GENERATOR	PURPOSE
$j:k$	A range: the integers from j to k
$j\#n$	n consecutive integers starting from j
$\{0, 1, 4, 9\}$	The elements of a set or list
$a.indices$	The indices of an array a

For example, the arguments to a function or method (or the operands of an operator) may all be evaluated in parallel. If the order of evaluation is important, the arguments must be explicitly ordered as statements in a block. Similarly, the iterations of a `for` loop may be executed in parallel; operations in different iterations may be freely interleaved.

Thus, the `for` loop...

```
for i <- 1:5 do
  print(i " ")
  print(i " ")
end
```

... prints each integer from 1 to 5 twice, but they may occur in any order, as in:

```
2 1 3 3 5 2 5 4 1 4
```

If the loop must be executed sequentially, the programmer must explicitly say so, as in:

```
for i <- sequential(1:5) do
  print(i " ")
  print(i " ")
end
```

Parallelism in `for` loops is controlled by *generators*, which produce the values that the loop iterates over. In the examples above, the generator `1:5` generates the integers from 1 to 5 in parallel, while the generator `sequential(1:5)` generates the integers in order. Generators, like most types, are provided by libraries and include aggregate types such as arrays and sets. Some common generators are shown in *Table One*. All these generators generate values in parallel.

Generators are also used in *comprehensions* and other *reductions*, which compute some function of each of the generated values and combines the results into a single value, as in:

```
{ i^2 | i <- 1:5 }
PRODUCT [i <- 1:n] i
```

```
C[i,k] = SUM[j <- 1:n] A[i,j] B[j,k], i <-
1:m, k <- 1:p
```

A naive implementation of a reduction may compute the individual results in parallel, but serialize the combining operations. If, as in all the examples above, the combining operation is associative and commutative, or it has other nice properties, it is possible to do much better. This is especially important for scientific programming, in which such reductions are common. The Fortress libraries support many different reductions, depending on the properties satisfied by the operation.

Fortress specifies when evaluation may proceed in parallel (or rather, when it must not), but does not require it to do so. If there are no spare processors, for example, or distributing the computation is more expensive than doing it, or for any other reason, an implementation may execute parallel tasks sequentially, and in any order. Thus, a programmer should allow as much parallelism as possible, giving the implementation maximal flexibility to execute the program efficiently. Using a technique called *work stealing*, an implementation can effectively distribute parallel tasks to multiple processors with minimal overhead.

Parallel execution of tasks accessing shared memory can result in surprising behavior due to race conditions. In existing languages, these are typically avoided using locks, which limit the parallelism possible and introduce a host of software engineering problems. To avoid these problems, Fortress supports transactional memory through the use of *atomic* expressions. An atomic expression appears to be executed in isolation: the evaluation of an atomic expression never appears to be interleaved with operations due to other tasks. Thus,

Fortress is implicitly parallel and provides constructs to serialize execution when necessary. A programmer should allow as much parallelism as possible.

other threads never observe a partially-completed atomic expression. Atomic expressions often allow greater parallelism than locks through the use of speculative execution, and free the programmer from the burden of managing locks correctly.

On large machines, another important factor is data distribution. The cost of access to memory is extremely nonuniform, so it is crucial to place data “near” to the processor that uses it. To manage this, Fortress provides *distributions*, which specify “regions” of the machine where data resides and computation occurs. These regions are arranged in a hierarchy that abstractly represents the relative cost of access: computation can cheaply access data in its own or nearby regions. As with parallelism, distributions are specified by generators, so that programmers need not think about data locality unless they design their own distributions, which are provided in libraries.

CATAclySMIC

...what we are seeing is not a gradual shift but a cataclysmic shift from the sequential world to one in which every processor is parallel. In a small number of years, if your language does not support parallelism, that language will just wither and die.

— John Mellor-Crummey, Rice University
(*Computerworld*, March 12, 2007)

Distributions are only advisory, but are a way for library writers to provide guidance to implementations as to how data should be distributed, a task that compilers have not been able to do very well thus far.

Fortress implementations are likely to aggressively use both static and dynamic compilation to improve performance, and the deployment model of Fortress is designed to support this. For example, Fortress runs in a managed environment, and thus benefits from automatic memory management (garbage collection) and just-in-time compilation. On the other hand, Fortress limits visibility into components via application programming interfaces (APIs), permitting a static compiler to optimize code within a component, as long as it preserves the behavior observable through its APIs. Furthermore, Fortress statically links components, enabling aggressive cross-component optimization at link time.

Mathematical Notation

Perhaps the most striking difference between Fortress and most existing programming languages is its syntax. As much as possible, Fortress syntax resembles the standard mathematical notation used by scientists. Fortress includes rich support for entering and displaying Unicode characters.

For example, Fortress allows the use of Greek letters (and many other character sets) as identifiers, braces to enclose sets, summation notation, and hundreds of different operators, including the use of simple juxtaposition to denote multiplication (as in $x y$) or function application (as in $\sin x$), depending on the types of the operands. Furthermore, Fortress specifies rules for rendering programs according to standard mathematical conventions, and for entering programs easily with a standard keyboard.

For instance, the righthand operand of the exponentiation operators is rendered as a superscript, and indices of matrix element are rendered as subscripts. Also, identifiers are rendered according to rules that allow programmers to use variables “decorated” with mathematical glyphs such as primes, “hats,” and overbars.

Using mathematical notation reduces the overhead for scientists to learn Fortress, because it looks like the equations already used in publications or a whiteboard. Ideally, programs

that look like the equations can reduce errors introduced in “translating” equations into program code, and enable scientists to more quickly find any errors that are introduced. Also, mathematical notation has developed over hundreds of years, and at its best, compactly and effectively expresses what needs to be computed, rather than how to compute it efficiently. The latter task is handled by libraries that implement the operations used in the equations, and are written by programmers who are experts in efficient algorithms, rather than scientists who are expert in a discipline.

As an example of Fortress code, *Listing One* shows a function that performs a matrix/vector “conjugate gradient” calculation (based on the NAS” CG” conjugate gradient benchmark for parallel algorithms). The function *conjGrad* takes a matrix and a vector, performs a number of calculations, and returns two values: a result vector and a measure of the numerical error. Most of the calculations either multiply a vector by a constant and add the result to another vector, or calculate the dot product of two vectors. In one place, however, a vector is multiplied by the given matrix; this is the operation that takes the most time and, fortunately, provides the greatest opportunities for speedup through parallel computation.

In Fortress, the function *conjGrad* has not only two value parameters *A* and *x*, but also four static parameters *Elt*, *N*, *Mat*, and *Vec*. The parameter *N* may be any natural number (non-negative integer). The other three static parameters are types; *Elt* may be any numeric type, *Mat* may be any type that is an *N*-by-*N* matrix with elements of type *Elt*; and *Vec* may be any type that is a length-*N* vector with elements of type *Elt*.

LISTING ONE: A function that performs a matrix/vector “conjugate gradient” calculation

```
conjGrad[Elt extends Number, nat N,
  Mat extends EltN×N,
  Vec extends EltN]
  (A: Mat, x: Vec): (Vec, Elt) = do
  cgit_max = 25
  z: Vec := 0
  r: Vec := x
  p: Vec := r
  ρ: Elt := r · r
  for j ← seq(1: cgit_max) do
    q = A p
    α = ρ / (p · q)
    z := z + α p
    r := r - α q
    ρ₀ = ρ
    ρ := r · r
    β = ρ / ρ₀
    p := r + β p
  end
  (z, ||x - A z||)
end
```

Because the length *N* of the vector is a parameter, *conjGrad* may be used with vectors (and matrices) of different sizes; because *N* is a static parameter, the Fortress compiler can check at compile time that the matrix is square and that its size matches the length of the vector.

One can specify all the type information explicitly when calling *conjGrad* on, for example, a 1000-by-1000 matrix *B* and a length-1000 vector *y*:

```
(z: RR641000, norm: RR64) =
  conjGrad[\RR64, 1000, RR64(1000 BY 1000),
  RR641000\] (B, x)
```

But that’s usually unpleasantly verbose. If the programmer omits the static type arguments...

```
(z, norm) = conjGrad(B, x)
```

... the Fortress implementation infers the necessary type information from the value arguments *B* and *x*. This is typical of Fortress coding style: programmers of library functions pay careful attention to type information so that application programmers need not worry much about types. As a result, type parameters appear extensively in library code, but are typically omitted and inferred in application code.

Fortress distinguishes between *defining* variables and *assigning* new values to variables. If you use a simple equals sign (=) to give a variable a value, that defines the variable and gives it a value, and that value cannot change; the variable is read-only, and the type of the variable is the type of the value. In the *conjGrad* example, *q* is defined to be equal to *A p*; for the rest of that block, *q* cannot change. (Of course, when the loop iteration ends, the *do* block completes and *q* becomes undefined; the next loop iteration starts a new execution of the *do* block and therefore creates a new definition for *q*.)

To define a variable that can be updated, you must use := instead of =, and you must specify the type of the variable explicitly. Then := can be used as an assignment operator to update the variable. In the *conjGrad* example, *z* is a vector

LISTING TWO: Part of the implementation of a “dense matrix” object

```
object DenseMatrix[Elt extends Number, nat M, nat N]
  (c: Elt[1: M, 1: N])
  extends EltM×N
  (* Multiply this matrix by a vector *)
  opr juxtaposition (v: EltN): EltM = do
    rowsum(i: Index) = ∑j=1:N cij vj
    DenseVector[i ↦ rowsum(i) | i ← 1: M]
  end
  ...
end
```

variable that is initially zero but can be updated, and indeed it is updated in the loop body. Unlike q , the value of z persists from one iteration of the loop to the next. (By the way, the loop is sequential because the function *seq* converts the generator `1:25` into a sequential generator that guarantees to perform loop iterations in ascending order. Without this use of *seq*, the loop iterations may be performed in parallel if there are extra processors available.)

There is no special `return` keyword; the value of a function is provided by the expression after the latest `=` in the definition, and the value of a `do` block is the last expression in the block. So the value of *conjGrad* is a 2-tuple, that is, a pair whose first item is the vector z and whose second item is the result of the matrix norm expression.

Whenever possible, Fortress implements a proposed language feature in a library rather than building it into the compiler. This allows the language to grow.

There is no explicit multiplication operator for multiplying a matrix by a vector; as in mathematics, one simply writes a juxtaposition of the matrix and the vector: Ax . But even though there is no symbol in the code, nevertheless juxtaposition is an operator just as real as `+` and `/`, and user-written library code can provide new definitions for this juxtaposition operator.

Listing Two shows part of the implementation of a “dense matrix” object; it represents a general M -by- N matrix with coefficients of type *Elt* as a two-dimensional M -by- N array of values of type *Elt*. (Note that “array” and “matrix” are distinct concepts in Fortress; a matrix is, among other things, a linear transformation on vectors, and has appropriate methods related to this purpose, but not every two-dimensional array

is a linear transformation on vectors.) The *DenseMatrix* object defines a method for the juxtaposition operator where the left-hand operand is the dense matrix and the right-hand operand is a vector. It uses a *local function* called *rowsum* that uses row i of the matrix to compute the value for one position of the result vector. The result of multiplying the matrix by the vector is a *DenseVector* object, created by providing a one-dimensional array of size M such the value of element i is *rowsum* (i). There is ample opportunity for parallel computation, not only because each row sum can be computed independently, but because the values to be summed can be computed independently and parallel summation algorithms can be used (for example, by organizing the summands as a binary tree).

There is more than one way to represent a matrix, of course, and in practice the matrix given to *conjGrad* is typically sparse.

Listing Three shows part of the implementation of a “sparse matrix” object; it represents a general M -by- N matrix with coefficients of type *Elt* using three one-dimensional arrays c , k , and w . The array c contains the nonzero elements of the matrix, grouped by row; the corresponding array k contains the column number for each nonzero element. The array w specifies how the elements of c (and k) are grouped; the elements in row i are stored in positions $w[i-1]$ through $w[i]-1$ of c . The arrays c and k should be the same length, but to keep the example simple, the code omits enforcing that constraint statically. (The indexing of vectors and matrices starts at 1, but the code uses index c and k starting at 0.)

The *SparseMatrix* object also defines a method for the juxtaposition operator. It is similar to the juxtaposition method for *DenseMatrix*, differing only in the details of how *rowsum* is computed, and provides similar opportunities for parallel computation. However, it may be much faster because it avoids the cost of multiplying vector elements by zero elements of the matrix.

The *conjGrad* code need not know anything about the specific matrix representation. The details of the matrix representation are hidden in the specification of the matrix objects, and the same implementation of *conjGrad* may be used with any object that implements the relevant matrix trait. This is object-oriented programming at its best. Other object-oriented languages such as C++ can provide this benefit, but Fortress integrates it with the use of familiar mathematical notation.

The Fortress Type System

Fortress is a multi-paradigm language: it supports many programming styles with a fairly expressive type system. It is mainly an object-oriented programming language, but also provides support for a functional style.

The Fortress type system is based on traits, a feature simi-

LISTING THREE: Part of the implementation of a “sparse matrix” object

```
object SparseMatrix[Elt extends Number, nat M, nat N]
  (c: Elt[0:], k: Index[0:], w: Index[1:N])
  extends EltM×N
  (* Multiply this matrix by a vector *)
  opr juxtaposition (v: EltN): EltM = do
    rowsum(i: Index) = do
      lo = if i = 1 then 0 else wi-1 end
      hi = wi - 1
      ∑j=lo:hi cj v[kj]
    end
    DenseVector[i ↦ rowsum(i) | i ← 1: M]
  end
  ...
end
```

lar to interfaces in the Java programming language, except that traits can contain code (but no fields). Traits in Fortress provide multiple implementation inheritance for better code sharing and better factoring. Fortress provides first-class functions and methods with multiple dispatch semantics. Functions and methods may be overloaded and the Fortress type system guarantees that there are no undefined nor ambiguous calls at run time.

Fortress also supports array-oriented “Fortran” style, which has been used in most of high-end scientific applications. Arrays are one kind of object supplied by Fortress libraries. While most scientific applications have been written in the Fortran style, array-oriented side effects may not scale well to petascale machines. Functional style may scale better and may provide more productivity for some applications because of the absence of side effects. Fortress keeps options open by providing both styles in libraries rather than wiring them into the compiler.

The Fortress type system is very expressive. It cleanly integrates nominal subtyping and generic types using “where” clauses. Where clauses express very general type constraints such as trait extensions only under certain conditions, relationships between types, and relationships between numeric values. Covariance and contravariance, which usually require additional language constructs, can be described using where clauses. A trait can take other types, numbers, booleans, and operator symbols as arguments. The type system can capture what is usually computed by data-flow analysis as type information. Unlike systems with type-erasure semantics, Fortress retains type information at run time to allow type-dependent operations on generic types, which is critical for expressing many common patterns in object-oriented programming.

This expressive power comes at a cost: the Fortress type system is complicated. However, application programmers need not understand the complicated type system. Rather, Fortress supports the rich type system for library writers to provide easy ways for application programmers to write Fortress programs. For the application programmers to use arrays with mathematical syntax without worrying about parallelizing the code, the library writers must have substantial flexibility and control over both syntax and semantics of the language, provided by the type system.

Growing a Language

Fortress is based on the following design strategy:

Whenever possible, implement a proposed language feature in a library rather than building it into the compiler.

That is, rather than add application-specific features to the language, Fortress provides support for programmers to implement those features in libraries. In other words, Fortress minimizes “compiler magic,” and instead makes that “magic” avail-

able to library writers. Thus, Fortress consists of a relatively small “core” and an extensive set of libraries, some of which implement such basic types as 32-bit integers and such basic constructs as `for` loops. Features in the core of Fortress are designed to support the writing of libraries more than the direct writing of applications.

This strategy supports the desire to *grow* the language, rather than *build* it. Because features are provided by libraries rather than built into the compiler, programmers can more easily customize and improve Fortress, and even extend the language by adding new features, making it easier to adapt Fortress as technology and programmers’ needs evolve.

For this approach to work, library writers must have substantial flexibility and control over both syntax and semantics, which Fortress supports with a highly-expressive type system. In addition, Fortress exposes structures that are typically internal to the compiler, such as the abstract syntax tree (AST) that result from parsing a program. These may be used to do program analyses and optimizations that would be done by the compiler in most languages, but must be done by libraries in Fortress.

By designing the language for growth, it can be extended via community participation and development. This is crucial to the success of Fortress, because there is a wide range of specialized libraries, especially for scientific computing, and one group is by no means best qualified to get them all right. Hence, Fortress is open source. In addition to developing libraries, or collaborating on the language implementation, the community is helping to build IDEs, analysis tools, and other artifacts to help Fortress programmers, exploiting the exposed Fortress abstract syntax tree to work cooperatively.

Eric Allen (PhD, Rice University 2003) is a Staff Engineer at Sun Microsystems, Inc. He is concerned with the design, formal analysis, and implementation of new language features to increase program robustness and programmer productivity. David Chase (PhD, Rice University 1988) is a Senior Staff Engineer at Sun Microsystems, Inc. He has worked on Fortress since 2003. Christine H. Flood (BS, Pennsylvania State University 1986) is a Member of the Technical Staff at Sun Microsystems, Inc. At Sun, she has worked on various JVM implementation issues. Jan-Willem Maessen (PhD, MIT 2002) joined Sun Microsystems in 2003 when the Programming Languages Research Group formed. Victor Luchangco (Sc.D, MIT 2001) is a member of the Scalable Synchronization Research Group and the Programming Languages Research Group of Sun Microsystems Laboratories. Suk-young Ryu (PhD, KAIST, 2001) is a Member of Technical Staff in Sun Microsystems Laboratories, where she works on formally designing and developing the Fortress programming language. Guy L. Steele Jr. (PhD, MIT, 1980) is a Sun Fellow at Sun Microsystems, Inc. In 1975 he co-invented the Scheme programming language at MIT with Prof. Gerald Jay Sussman.