A Comparative Study of Fortran With C++ and Java Using an Object-Oriented Ray Tracing Application

Shahid Alam
Department of Computer Science
University of Victoria, BC V8P 5C2
E-mail: salam@cs.uvic.ca

1 Introduction and Motivation

One of the most important aspects of a programming language is its ability to provide higher levels of abstractions to the programmer. A programmer can define abstract data types such as: types with extensions in Fortran, interfaces in Java and classes in C++.

According to the Tiobe Language Popularity Index [12] the three most popular programming languages in any category (including general-purpose, compiled, script and other) are Java, C and C++ in that order. Fortran is listed at number 33 in the index. Another Language Popularity Index [32] lists the three most popular programming languages in the general-purpose and compiled category as C, Java and C++ in that order. Fortran is listed at number 12 in the index. These indexes are based on searching the web (Google, MSN and Yahoo etc) with certain phrases that contain the language names and counting the number of hits returned. These indexes show that Fortran is not very popular as a general-purpose programming language. An article, published by ACM Queue, The Ideal HPC Programming Language [29], argues that Fortran is still one of the popular and primary languages in high performance computing (HPC).

As COBOL is popular in large corporate business data centers and C in embedded and operating systems, Fortran [11] has been in use and popular for over 50 years in scientific and engineering communities. Most of the compute intensive tasks such as weather and climate modeling, computational chemistry and physics, and others are programmed in Fortran. Most of the SPEC CPU2006 floating point benchmarks [18] are implemented in Fortran.

Java [26] compared to Fortran and C++, is a relatively new language and was first released in 1995. It is mostly used for application development, specifically for the web. It has a simpler object model and fewer low level facilities than C++. Applications written in Java can run on any machine with a JVM (Java virtual machine). This makes it a "write once, run anywhere" language and hence a popular choice among the software developers.

C++ [9] is mostly used to write system and application software, high performance server and client applications, and video games. C++ is an enhancement to C and was initially called C with classes. The previous C++ standard [7] was published in 1998 and revised in 2003. The recent standard [9] known as C++11 was passed in 2011.

There is a large code base including libraries of Fortran that exist and are being developed by scientists and engineers for specialized applications. Efficient development and maintenance of this code is key to the
success of the Fortran language. To improve these software development processes, Fortran has gone through some major and minor extensions in 2003 and 2008. The most recent standard known as Fortran 2008\textsuperscript{11} with these extensions was passed in 2010.

Fortran and C++ both have been officially approved by the ISO standards committee. Java does not have an official standard approved by any of the standards committee but has achieved a dominant position by public acceptance and market forces. This paper presents a comparative study to evaluate and compare Fortran 2008 with Java and C++ making a case on behalf of the software developers from the scientific and engineering communities that Fortran is still relevant, and to highlight some of it’s significant advantages. We are interested in answering the questions: (1) How much have the new extensions made Fortran comparable to Java and C++? (2) What are some of the similarities and differences in supporting features like: Templates, object constructors and destructors, abstract data types and dynamic binding?

These three languages are traditional languages and are being used for developing large software applications. The program for comparison should be large enough to cover most of the features of the programming language. A simple program will not reveal all the similarities and differences among the compared languages, and a very large program can make the comparison complex and the results unusable.

A basic ray tracer is implemented to compare the three languages. The ray tracing application implemented in this paper is neither complex nor simple, but is practical and complete enough that it has been used to generate molecular model animations (visualizations). The ray tracer is an object-oriented application and can either render one image or more than one images (animation). Depending on the complexity of the animation the rendering can take a lot of CPU cycles. We also give a comparison of the runtime of the ray tracing application for each language and highlight some of the major differences of object handling and processing. While generating animation the ray tracer processes a lot of images and writes them to the disk. A correct measurement of the size and careful inspection of the code will give us an insight into the complexity, quality of the code, similarities and differences in each language.

Tables 1 and 2 compare some of the basic and advance features of the three languages. Comparison of all the features of these languages is out of the scope of this paper. These tables cover only some minor (basic) and major (advanced) features of these languages.

2 Design of the Ray Tracer

Figure\textsuperscript{1a} shows a flow chart of the ray tracing application. The complete source code for the ray tracer in all the three languages is available online @\url{http://www.cs.uvic.ca/~salam/raytracer/raytracer.html}. The ray tracer implemented here only renders spheres and planes. The rendering of images (scene) is done by ray tracing each ray from the camera. The number of rays depends on the height and width of the scene. A scene file is used to define spheres, planes, lights, materials and paths in the scene. The ray tracer generates animation by using the object Path defined in the scene file. A Path is attached to an object in the scene file to generate animation for that object. The Path contains information of an elliptic path. The ray tracer uses this information to compute the positions around the elliptic path. Then it generates images at each position. These images together create an animation. More details about the scene file, rendering of animation and the
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Programming Paradigm</strong></td>
<td>Procedural</td>
<td>Procedural</td>
<td>Procedural</td>
</tr>
<tr>
<td></td>
<td>Object oriented</td>
<td>Object oriented</td>
<td>Object oriented</td>
</tr>
<tr>
<td><strong>Data types</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td>integer (kind = n)</td>
<td>char (8 bit)</td>
<td>byte (8 bit)</td>
</tr>
<tr>
<td></td>
<td>short (16 bit)</td>
<td>long (32 bit)</td>
<td>char (16 bit)</td>
</tr>
<tr>
<td></td>
<td>long long (32 bit)</td>
<td>int (word)</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td></td>
<td>real (kind = n)</td>
<td>float (single precision)</td>
<td>float (single precision)</td>
</tr>
<tr>
<td></td>
<td>double precision</td>
<td>double (double precision)</td>
<td>double (double precision)</td>
</tr>
<tr>
<td>Complex numbers</td>
<td>complex (kind = n)</td>
<td>std::complex&lt;float&gt;</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>double precision</td>
<td>std::complex&lt;double&gt;</td>
<td></td>
</tr>
<tr>
<td><strong>Character</strong></td>
<td>character (len = *)</td>
<td>char</td>
<td>char</td>
</tr>
<tr>
<td><strong>String</strong></td>
<td>character (len = :)</td>
<td>std::string</td>
<td>String</td>
</tr>
<tr>
<td><strong>Boolean</strong></td>
<td>logical (kind = n)</td>
<td>bool</td>
<td>boolean</td>
</tr>
<tr>
<td><strong>Object</strong></td>
<td>class (*)</td>
<td>void *</td>
<td>Object</td>
</tr>
<tr>
<td><strong>Array</strong></td>
<td>type::name(size)</td>
<td>std::array&lt;type, size&gt;</td>
<td>type[]</td>
</tr>
<tr>
<td><strong>Control flow</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Conditional</strong></td>
<td>if (condition) then statement</td>
<td>if (condition) statement</td>
<td>Same as C++</td>
</tr>
<tr>
<td></td>
<td>else statement</td>
<td>else statement</td>
<td></td>
</tr>
<tr>
<td></td>
<td>endif</td>
<td>switch (condition)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>select case (case-expr)</td>
<td>case const-expr:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>case (const-expr)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>end select</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Loops</strong></td>
<td>do do-var = int-expr, int-expr</td>
<td>while (condition) statement</td>
<td>Same as C++</td>
</tr>
<tr>
<td></td>
<td>do-block</td>
<td>do statement</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>while (expression);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>for (init; final; increment) statement</td>
<td></td>
</tr>
<tr>
<td></td>
<td>end do</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Exceptions</strong></td>
<td>x</td>
<td>try statement</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>catch (1E) statement</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>throw E;</td>
<td></td>
</tr>
<tr>
<td><strong>Enumerations</strong></td>
<td>1 enum, bind(C)</td>
<td>3 enum ViewType {</td>
<td>2 public enum ViewType {</td>
</tr>
<tr>
<td></td>
<td>enumerator :: ORTHOGRAPHIC = 0</td>
<td>ORTHOGRAPHIC,</td>
<td>ORTHOGRAPHIC,</td>
</tr>
<tr>
<td></td>
<td>enumerator :: PERSPECTIVE = 1</td>
<td>PERSPECTIVE</td>
<td>PERSPECTIVE</td>
</tr>
</tbody>
</table>
|                         | endenum      | | };
| **Methods**             | subroutine sub(parameters) | void sub(parameters) | Same as C++ |
|                         | statement    | { statement } | |
|                         | end subroutine | | |
|                         | type function func(parameters) | type func(parameters) | |
|                         | statement func = value | { statement return value; } | |
|                         | end function | | |
| **Memory management**   | Manual. Some keywords used are: | Manual. Some keywords used are: | Automatic by the Garbage |
|                         | allocate, deallocate | new, delete | Collector and can be |
|                         | Objects allocated | Objects allocated | triggered manually. |
|                         | on heap and stack | on heap and stack. | Objects allocated on heap. |
|                         | 4 finalizer method | Class destructor | |
|                         | | | 5 finalizer method |
| **Runtime**             | Compiled to machine code and executed by the operating system. Prone to low level errors. Bounds checking for arrays through compiler options. | Compiled to machine code and executed by the operating system. Prone to low level errors. Bounds checking for arrays through library. | Compiled to bytecode and executed by the Java virtual machine. |
|                         | | | Low level errors cannot occur. Automatic bounds checking for arrays. |

1 E = exception
2 Example shown from the source code of the ray tracing application.
3 In the ray tracing application we used #define in C++ and final variables in Java.
4 Fortran provides a final method as finalizer. More details are given in Table 2.
5 A special method that is executed when an object is garbage collected. Unlike Fortran finalizer and C++ class destructor the Java specification does not guarantee when it will be called.
6 We have only featured the normal runtime behavior of the three languages.

Table 1: Comparison of Some of the Basic Features of Fortran, C++ and Java
## Table 2: Comparison of Some of the Advance Features of Fortran, C++ and Java; Examples shown are from the ray tracing application

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic Types</td>
<td>Parametrized derived types</td>
<td>Templates</td>
<td>Generics</td>
</tr>
<tr>
<td>type ListNode(k)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer, kind := k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>real(kind=k) :: data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>type(ListNode(k)), pointer :: next, prev</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end type ListNode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>type(ListNode(L)) U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>type(ListNode(N)) B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Namespace</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Operator Overloading</td>
<td></td>
<td>Vector &amp; Vector : operator=(Vector &amp;v)</td>
<td>x</td>
</tr>
<tr>
<td>interface assignment(=)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end interface</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>subroutine assign(V1, V2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end subroutine</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inner Classes</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Local Classes</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Final Classes</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Final Methods</td>
<td>final : final-subroutine-name-list</td>
<td>A class destructor: ~class-name()</td>
<td>final return-type method-name ([argument-list]) { method-body }</td>
</tr>
<tr>
<td></td>
<td>Automatically invoked when the object is finalised</td>
<td>Automatically invoked when the object is destroyed</td>
<td>Prevent subclasses from overriding it and is different than the finalizer method</td>
</tr>
<tr>
<td>Final Variables</td>
<td>type, parameter :: variable-name = const-expression</td>
<td>const type variable-name = const-expression</td>
<td>final type variable-name = const-expression</td>
</tr>
<tr>
<td>Abstract Classes</td>
<td>type, public, abstract :: Shape</td>
<td>class Shape</td>
<td>public abstract class Shape</td>
</tr>
<tr>
<td></td>
<td>procedure(Hit(d), pass, deferred :: Hit</td>
<td>virtual bool Hit(Ray &amp;ray, double &amp;t) = 0;</td>
<td>public abstract boolean Hit(Ray ray, Tee t);</td>
</tr>
<tr>
<td></td>
<td>end type Shape</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>abstract interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>logical function Hit(d)(S, R, t)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><img src="#" alt="For full example see Fig 1" /></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>end function Hit(d)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual Functions</td>
<td>As shown above</td>
<td>As shown above</td>
<td>All non-static methods</td>
</tr>
<tr>
<td>Interfaces</td>
<td>Abstract and procedure interfaces as shown above</td>
<td>An abstract class with pure virtual function(s) as shown above</td>
<td>interface Image</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>interface Image</td>
<td>public void Read(String filename);</td>
<td>public void Write(String filename);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>public void SetColor(int r, int g, int b);</td>
</tr>
<tr>
<td>Anonymous Classes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Only unions: union { member-specification }</td>
<td>Special case of an inner class to a method and is implicitly final: new class-name ([argument-list]) class body</td>
<td></td>
</tr>
<tr>
<td>Debugging</td>
<td>stop and error stop statements</td>
<td>Diagnostics standard library</td>
<td>Java platform debugger architecture</td>
</tr>
<tr>
<td></td>
<td>Compiler specific flags e.g. funtrace [22]</td>
<td>&lt;stbtrace&gt;, &lt;cassert&gt;</td>
<td>JDI and JVMTI [14]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;cerrno&gt; and &lt;system_error&gt;</td>
<td></td>
</tr>
</tbody>
</table>
design of the ray tracer is given in [5].

The function $RayTrace()$ is compute intensive. The pseudocode of this function is listed in Figure 1b. The computation intensity depends on the size of the scene, number of objects in the scene, and the number of reflections and refractions by the ray. At each iteration the function $RayTrace()$ checks all the objects in the scene for a hit by the ray. If a ray hits an object the shading, shadow, reflection and transparency/refraction are computed, which are then used to compute the color of the pixel at the hit position. The reflection and refraction is performed by recursively iterating on the reflected and refracted rays.

---

Figure 1: Ray Tracer Flow Chart and the Function $RayTrace$
3 Applying Basic Object-Oriented Software Metrics to the Ray Tracer

Tables 3 and 4 show size-level and class-level metrics [2, 31] for the ray tracing application. We have only selected basics of these metrics to represent complexity and quality of the object-oriented features of the ray tracing application for comparing the three languages. The purpose of applying these metrics is to highlight the differences and similarities among these languages such as the same depth of inheritance achieved in all three languages, and that there are a greater number of classes in Fortran and Java than C++. Overall
there are more similarities than differences that proves our conjecture, that Fortran as an object-oriented language is equivalent to one of the two most popular object-oriented languages Java and C++. In the next Sections we will see that there are some differences that need to be improved to make Fortran a more relevant object-oriented language.

3.1 Size-Level Metrics

There are 19 in Fortran, 17 in C++ and 24 classes in Java as shown in the Size-Level Metrics in Table 3. This difference is because there are no struct or union types in Fortran and Java. We used types in Fortran and classes in Java to imitate type struct, shown in Figure 2 as type(Color) and class Color. Also in Java the global definitions are defined in a class and the parameters for the recursive function RayTrace() are passed as classes. Java has more in total, but less mean number of attributes (member variables of a class) than Fortran and C++ because of the greater number of classes in Java. This confirms that Java is a pure object-oriented language and everything is an object in Java. In Fortran and C++ the programmer is not bound to use objects.

Class Constructor: In C++ and Java the class constructor has a much simpler declaration than Fortran. In Figure 2 the Fortran class TGA’s constructor is declared as interface TGA with procedure init_TGA().

Class Destructor: As mentioned before there is no class destructor in Java and the deallocation of objects is taken care of by the garbage collector. C++ uses the symbol ~ to declare the class destructor as shown TGA::~TGA() in the Figure. In Fortran a destructor for a class is declared using the keyword final and is implemented using the procedure final_TGA() in the Figure. The dynamic memory allocated for the object Color in Figure 2 is deallocated, in Java by the garbage collector (controlled by the JVM - Java Virtual Machine - and not by the application), in C++ when the statement delete(tga) is called and in Fortran just before the end program example statement is reached.

As shown in the Size-Level Metrics graph in Table 3 the total number of methods in the Fortran source code of the ray tracing application is less than the total number of methods in the C++ and the Java source code. Because the init (Class constructor) and the final (Class destructor) methods were not implemented for all the Fortran classes. The gfortran compiler version 4.7 used in this paper supports quite a few of these Fortran 2003 and 2008 extensions but does not support all of them. The class constructor and destructor are not supported by the gfortran compiler as declared in Figure 2. For a complete list of Fortran compilers and their support for the Fortran 2003 and 2008 standards the reader is referred to.

<table>
<thead>
<tr>
<th></th>
<th>Fortran</th>
<th>C++</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>TM</td>
<td>151</td>
<td>165</td>
<td>154</td>
</tr>
<tr>
<td>TA</td>
<td>78</td>
<td>82</td>
<td>101</td>
</tr>
<tr>
<td>TC</td>
<td>19</td>
<td>17</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 3: Size-Level Metrics for the Ray Tracing Application in Fortran, C++ and Java

TM, TA and TC are the total number of methods, attributes and classes respectively.
3.2 Class-Level Metrics

An explanation and discussion of the Class-Level Metrics shown in Table 4 follows. The Min number for each of these metrics is 0 and the Mean is the arithmetic mean:

```fortran
  type, public, abstract :: Shape
  contains
    procedure(SetPosition_d), pass, deferred :: SetPosition
    procedure(GetPosition_d), pass, deferred :: GetPosition
    procedure(Hit_d), pass, deferred :: Hit
  end type Shape
  abstract interface
    subroutine SetPosition_d(S, X, Y, Z)
      import Shape
      class(Shape), intent(inout) :: S
      double precision, intent(in) :: X, Y, Z
    end subroutine SetPosition_d
    type(Vector) function GetPosition_d(S)
      import Shape, Vector
      class(Shape), intent(in) :: S
    end function GetPosition_d
    logical function Hit_d(S, R, t)
      import Shape, Ray
      class(Shape), intent(in) :: S
      class(Ray), intent(inout) :: R
      double precision, intent(inout) :: t
    end function Hit_d
  end interface

  type, public, extends(Shape) :: Sphere
  contains
    procedure, pass :: SetPosition=>SetPosition_SPHERE
    procedure, pass :: GetPosition=>GetPosition_SPHERE
    procedure, pass :: Hit=>Hit_SPHERE
  end type Sphere

  type, public, extends(Shape) :: Plane
  contains
    procedure, pass :: SetPosition=>SetPosition_PLANE
    procedure, pass :: GetPosition=>GetPosition_PLANE
    procedure, pass :: Hit=>Hit_PLANE
  end type Plane
```

Figure 3: Part of the Source Code of the Ray Tracer Classes Shape, Sphere and Plane in Fortran

- **NOA**: Number of attributes per class. An attribute represents the structural properties of a class and is defined as part of the declaration of the class. All three languages have almost the same mean number of attributes per class except Java, because of the use of classes in place of struct.

- **NOM**: Number of methods per class. A method is an operation upon an object and is defined as part of the declaration of a class. The Mean number of methods per class in the Fortran and the Java is less than the C++. As mentioned before there are more classes in the Fortran and the Java than the C++ implementation. These extra classes are user defined data types and have 0 number of methods.

- **DIT**: Depth of inheritance. This is the number of ancestor classes also called super classes to a sub class. In our ray tracing application there are two super classes: Shape ⇒ [Sphere and Plane] and Image ⇒ [TGA]. All the languages have exactly the same number of superclasses. The difference in Mean is
### Table 4: Class-Level Metrics for the Ray Tracing Application in Fortran, C++ and Java

<table>
<thead>
<tr>
<th>Metric</th>
<th>Fortran</th>
<th>C++</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
<td>Mean</td>
</tr>
<tr>
<td>DAC$^1$</td>
<td>0</td>
<td>8</td>
<td>2.4</td>
</tr>
<tr>
<td>DIT$^2$</td>
<td>0</td>
<td>1</td>
<td>0.16</td>
</tr>
<tr>
<td>NMO$^3$</td>
<td>0</td>
<td>3</td>
<td>0.5</td>
</tr>
<tr>
<td>NOA$^4$</td>
<td>0</td>
<td>28</td>
<td>4.7</td>
</tr>
<tr>
<td>NOM$^5$</td>
<td>0</td>
<td>26</td>
<td>8</td>
</tr>
</tbody>
</table>

1 Data abstraction coupling  
2 Depth of inheritance  
3 Number of methods overridden by a class  
4 Number of attributes per class  
5 Number of methods per class

because of the difference in total number of classes. Similar to Java, Fortran uses the keyword `extends` to inherit a super class. As shown in Figure 3, the super class `Shape` is inherited by the two sub classes `Sphere` and `Plane`. The inheritance in Fortran is very similar to C++ and Java. In Fortran the sub classes inherit all the attributes and methods of the super class. Out of the three languages only C++ supports multiple inheritance and Templates.

- **NMO**: Number of methods overridden by a class. This makes a class polymorphic also called *ad-hoc polymorphism* [33]. A method defined in a super class is implemented differently in each of the sub classes. In Fortran the keywords `abstract` and `deferred` are used to declare a virtual function. In our *ray tracing* application, as shown in Figure 3, the sub classes `Sphere` and `Plane` implement the three virtual functions `SetPosition()`, `GetPosition()` and `Hit()` declared in the super class `Shape`. All of the three languages provide facilities to abstract either a complete class or an individual function. One of the super classes, `Image`, is an interface. Three functions in the `Shape` class are declared abstract in all three languages, an example in Fortran is shown in Figure 3. These functions are implemented by the sub classes `Sphere` and `Plane`. The difference in Mean is due to the difference in the total number of classes.

- **DAC**: Data abstraction coupling. It is the ability to create new data types called abstract data types (ADTs). DAC is the number of ADTs defined in a class. Some of the ADTs in our *ray tracing* application are: `Vector`, `Ray` and `RGBColor`. Java is a pure object-oriented language and the programmer is bound to use objects (classes) for every user defined data structure. Therefore the *ray tracing* application implemented in Java has more data structures defined as classes (e.g: in the *ray tracing* application implemented in Java various `struct` types are defined as ADTs) and hence a bigger DAC number when compared to Fortran and C++.

We have discussed some of the differences and similarities in the three languages by looking at the source code and applying object-oriented metrics to the *ray tracing* application. These metrics and the discussion imply that Fortran is an equal value object-oriented language like C++ and Java. The syntax may be different but Fortran supports most of the object-oriented features present in C++ and Java. Some of the features like
Templates (Fortran Parametrized Derived Types and Java Generics are technically different than Templates as listed in Section 1) and multiple inheritance are not supported in Fortran for the same reason they are not supported in Java due to their runtime overhead and the complexities of implementing an optimizing compiler. Templates with many lines of code cannot be inlined and may incur runtime overhead. For details of, why multiple inheritance make things complex and increases the runtime overhead the reader is referred to [19]. C++ and Java are more concise and clear, but Fortran is more verbose and explicit.

4 Basic Software Metrics for the Ray Tracer

Figures 4 and 5 shows and Table 5 lists some of the basic software metrics for the ray tracing application, including the runtime and the size of the ray tracing application. The reason for including the size of the ray tracing application is to: show that Fortran is a verbose language and why it took more time to implement the application in Fortran. We have chosen four parameters to enumerate the size: number of files, number of bytes, words and lines of the source code. We believe all these parameters together give a correct measurement of the size of the code and also give an insight into the complexity of the code. The number of words and physical lines of source code also includes comments. These comments are almost the same for each language. We used the following timing functions, as recommended by all the three languages standard, in this paper to compute the CPU time: cpu_time() in Fortran (ref: Section 13.7.42 of [11]), clock() in C++ (ref: Section 7.23.2.1 of [8]) and nanoTime() in Java [15].

The runtime shown in Figures 4 and 5 is the CPU time in milliseconds to render each image for generating the animation. The scene files that were used to generate animations are listed in [5]. The top (Simple Scene Animation) and the bottom (Complex Scene Animation) animations contain 1296 and 1245 total number of images and, 5 and 56 total number of objects, respectively. The ray tracing application in Fortran, using the open source compiler gfortran shown in Figure 4, on average took 100% and 300% more time than the C++ application to render each image for the top and the bottom animations respectively. Java is the slowest when rendering animation for the Simple Scene but Fortran is 25% slower than Java when rendering the Complex Scene.

Our first thought was that Fortran is spending more time in file I/O than the other two languages. But further instrumentation at the function level of the source code shows that in the three languages compared in this paper the ray tracer spends most of the time, almost 90%, in the recursive function RayTrace(). Due to the object oriented features of the ray tracing application, this difference shows that these features are not implemented as efficiently in the gfortran as in the g++ compiler. Here we also want to point out a fact that Java is slower at startup and is shown in our experimental results in Figure 4. As gfortran is an open source free compiler, and to make a precise conclusion, we also present and compare the results using the IBM XL compilers in Figure 5.

The ray tracing application in Fortran, using the IBM XLF compiler shown in Figure 5 on average took 400% and 500% more time than the C++ application to render each image for the top and the bottom animations respectively. Java is the slowest when rendering animation for the Simple Scene, but Fortran is only 30% faster than Java in SUSE Linux and 1.6% slower in Red Hat Linux when rendering the Complex
Following experimental setup was used to generate the data shown here:

**Machine used:** Intel Core2 Quad CPU Q6700 4GB RAM; Dedicated for the experiment i.e. why the results are more uniform than shown in Figure 5.

**Compilers:**
- gfortran 4.7
- g++ 4.7
- Oracle Java 1.6.0.21

**Optimization Flags:** -O3 for gfortran and g++, and no flag used for Java

Figure 4: Runtime of the *Ray Tracing* Application (Using GNU Compilers), in Fortran, C++, and Java, for Rendering Images to Generate Animation.
(a) SUSE Linux Enterprise Server 11 SP1, 64 bit, Linux kernel: 2.6.32.12; Power7 16 processors 4GB RAM

(b) Red Hat Linux Enterprise Edition 5.6, 64 bit, Linux kernel: 2.6.18-238; Power6 16 processors 8GB RAM

Following experimental setup was used to generate the data shown here:

**Machines used** were not dedicated for the experiment. These machines are continuously used for running different experiments and tests at IBM. The reason for using these machines is to simulate the real life environment where the load is always unbalanced i.e. why the results are non-uniform compared to Figure 4.

**Compilers:** IBM XLF V13.1 [25], IBM XLC V 11.1 [26] and IBM J9 VM 1.6.0 [24]

**Optimization Flags:** -O3 for XLF and XLC, and no flag used for Java

Figure 5: Runtime of the *Ray Tracing Application* (Using IBM XL Compilers), in Fortran, C++ and Java, for Rendering Images to Generate Animation.
Scene. These results confirm that our conclusion above also applies to the IBM XL compilers. We provide an assessment of these and other results in the next Section. Further analysis of the source code at the statement level shows that as we increase the number of objects in the scene the Fortran application took more time to render each image. Based on this analysis of object processing in the three languages, we present another comparison in Figure 6 which shows the average runtime of the function RenderOneImage of the Ray Tracing application using a range of, from 5 - 400, objects. To emphasize and compare the processing of the objects in the scene we simplified the RayTrace function as shown in Figure 6.

Figure 6 shows that Fortran remains ahead of Java when the number of objects are 20 or less. After 20 as we double the number of objects so does the runtime of Fortran processing these objects. With the introduction of object-oriented features and to make Fortran relevant, competitive and successful in this area both the open source and the commercial compilers need to improve this particular area. One of the reasons it is not optimized in these two popular compilers is that either the Fortran programmers (engineers and scientists) are not using or the use is still immature of the object-oriented features of the language. These object-oriented features are still very new to the Fortran language and will take some time to become mature and stable among programmers and the tool developers. We provide more insight into this in Section 5. As mentioned before this is one of the motivations of this paper to increase the use of these features among the programmers and improve the available tools.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Fortran</th>
<th>C++</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Files</td>
<td>18</td>
<td>31</td>
<td>19</td>
</tr>
<tr>
<td>Size in Bytes</td>
<td>101554</td>
<td>74959</td>
<td>70965</td>
</tr>
<tr>
<td>Size in Words</td>
<td>11987</td>
<td>9287</td>
<td>8703</td>
</tr>
<tr>
<td>Physical Lines</td>
<td>3336</td>
<td>3254</td>
<td>2828</td>
</tr>
</tbody>
</table>

Table 5: Size Metrics for the Ray Tracing Application in Fortran, C++ and Java

Fortran has a total 18 files, 16 of which contain class definitions and 2 that contain global parameters. These 2 files can also be called global header files. C++ has a total of 16 header files and therefore has more files than Fortran and Java. In Java the file name and the class defined in the file should have the same name. There are a total of 24 classes in the Java ray tracer as shown in the Size-Level Metrics in Table 3 of which there are 5 inner classes, therefore there are only 19 files in Java. We mentioned Fortran as a verbose programming language which is confirmed by the larger sizes of the ray tracing application in Fortran listed in Table 5. It took more time to implement the application in Fortran than Java and C++. Java implementation took the least time. The size of the code also follow the same order as shown in Table 5.

4.1 Assessment

This section assesses the validity of the methodology used for the experiments and the results presented above. By using one application we provided one set of requirements for a homogeneous comparison. Hence the data set in one language provided the same properties as the data set in other languages. The experiments were carried out in a controlled environment. To increase the validity of the results we: provided data for four popular compilers including, GNU and IBM XL for Fortran and C++, and Oracle and IBM for Java; used
Following experimental setup was used to generate the data shown here:

**Machine used:** Power7 16 processors 4GB RAM; SUSE Linux Enterprise Server 11 SP1, 64 bit, Linux kernel: 2.6.32.12

**Compilers:** IBM XLF V13.1, IBM XLC V 11.1 and IBM J9 VM 1.6.0.

**Optimization Flags:** -O3 for XLF and XLC, and no flag used for Java.

To emphasize and compare the processing of the objects, the following simplified **RayTrace** function was used. We filtered the noise (File I/O, shading, reflection and refraction etc) from the CPU time. It was easy to make sure that this small part of the code is almost the same in all the three languages. To make the comparison more fair (specifically for Fortran), for iterating the objects in: Java we used an iterator to access the private objects; C++ we used an array of pointers to access the private objects; Fortran we directly access the public objects:

```fortran
logical function RayTrace(color, ray, depth)
    hit = 0
    for i = 1, length(objects)
        if (hit(objects[i], ray)) then
            object_hit = objects[i]
            hit = 1
        end if
    end for
    material = get_material(object_hit)
    if (hit == 0) then
        return false
    end if
    color = get_color(material)
    return true
end function RayTrace
```

**Figure 6:** Average Runtime of the Function **RenderOneImage** of the **Ray Tracing** Application, Using Different Number of Objects (planes, spheres etc in the scene), in Fortran, C++ and Java; The Average is Computed over 30 Rendered Images.
four different systems as shown in Figures 4 and 5 to carry out the experiments; instrumented and traced the source code to function level and presented the data for a range of objects as shown in Figures 6 and 7.

Same optimization level was used both for GNU and IBM XL compilers. Oracle and IBM Java were enabled for JIT (just in time) optimization. To our knowledge there is no method or technique available that guarantees that the optimizations applied during JIT optimization of a Java program will be the same as applied in static compilation of a Fortran or a C++ program.

Java was designed as both interpreted and compiled language. Here we want to mention and highlight the fact that Fortran was compiled and Java was interpreted. In the Ray Tracing application the change in the number of objects and various loop counters is dynamic, that makes it more suitable for dynamic optimization as is done by the JIT optimizer. In Section 5 we discuss the reasons why Java code was not compiled to native code for the purpose of comparing the performance.

The reasons for using the GNU compilers in addition to the IBM compilers are as follows: (1) All the GNU compilers (g++, gfortran and gcj) use the same back-end. This helped us in analyzing the program in Section 5. (2) There availability on most of the platforms even Windows, that is one of the platforms we used for testing. (3) They are as popular and used in desktops, embedded systems and servers as any other good compiler, and are adopted as the standard compilers in most modern Unix-like operating systems such as Linux.

To validate the timing functions, recommended by the each language standard and used in this paper, we timed a simple loop (implemented in Fortran, C++ and Java) with a large count, embedded inside other loops. We made sure that the statement inside the loop is not simple enough to be optimized by a smart compiler in a way that the statement never executes or executes less than the loop count. Here are the average timings in milliseconds for 30 iterations of this loop: C++ = 9380; Fortran = 9667; Java = 10016. These are the expected timings and indicate that we can safely use these timing functions.

To assess and validate our results, for the Ray Tracing application, shown in Figure 6 we carried out another experiment with a simple object-oriented application implemented in the three languages as shown in Figure 7. The loops in the main of each language are exactly the same as used above to validate the timing functions, except the one additional statement of setting the position of the object Sphere. The results in Figure 7 shows the Fortran slowing down as we increase the number of objects.

So far we have shown that Fortran lags when we increase the number of objects. In the next section we perform binary analysis and profiling of this simple object-oriented application to highlight and discuss where Fortran lacks in optimization of object handling and processing.

5 Binary Analysis And Profiling

Comparing the performance of programming languages is a non-trivial task. As mentioned previously the purpose of the experiments and the results shown in the graphs and the tables of this paper, is to compare the performance of these object-oriented languages for efficiently handling and processing the objects. In this section we highlight this particular area of the Fortran language and argue that it needs improvement by profiling the runtime and analyzing the assembly codes generated by the three compilers. We use GNU
**Program Main**

```fortran
integer, parameter :: NUM_SPHERES = 500
use SPHERE_
program main
  double precision :: COUNT
  real :: total_time
  real :: time_start, time_end
  type(Sphere), allocatable :: sphere_t(:)
  double precision, intent(in) :: X, Y, Z
  class(Sphere), intent(inout) :: S
  type, public, extends(Shape) :: Sphere
end interface
end module SPHERE_
end program main
```

**Subroutine SetPosition**

```
subroutine SetPosition_Sphere(S, X, Y, Z)
  import Shape
  double precision :: x, y, z
  class(Shape), intent(inout) :: S
  double precision, intent(in) :: X, Y, Z
  S%x = X    S%x = Y   S%x = Z
end subroutine SetPosition_Sphere
```

**Figure 7:** Average Runtime of the Loop in the Small Object-Oriented Application, Using Different Number of Objects (Spheres), in Fortran, C++ and Java; The Average is Computed over 30 Iterations.
compilers for C++ and Fortran and Oracle compiler for Java.

We profiled the simple object-oriented application discussed in Section 4.1 as shown in Figure 7 using the PIN tool [30]. PIN is a dynamic binary instrumentation tool, i.e., it performs instrumentation at runtime. We wrote a small PIN application to count the number of times a routine is executed (number of calls). Since the instrumentation is done at the binary level, it includes routines both from the application and from the dynamic libraries linked at the runtime. We also counted the total number of instructions executed in each of these routines. After examining the results we found that the Fortran application spent most of its time in the function (routine) Sphere::SetPosition as shown in Table 6. We also list the generated assembly and byte code for the function Sphere::SetPosition in Table 7.

<table>
<thead>
<tr>
<th>Language</th>
<th>Function</th>
<th>Image</th>
<th>Address</th>
<th>Calls</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>Sphere::SetPosition</td>
<td>simple</td>
<td>0x400A90</td>
<td>393216000</td>
<td>1572864000</td>
</tr>
<tr>
<td>Fortran</td>
<td>Sphere::SetPosition</td>
<td>simple</td>
<td>0x400950</td>
<td>393216000</td>
<td>3145728000</td>
</tr>
<tr>
<td>C++</td>
<td>main</td>
<td>simple</td>
<td>0x400830</td>
<td>1</td>
<td>5910828522</td>
</tr>
<tr>
<td>Fortran</td>
<td>main</td>
<td>simple</td>
<td>0x400970</td>
<td>1</td>
<td>9057340465</td>
</tr>
</tbody>
</table>

1 Image to which the function (routine) belongs, and can be the application or the library. Here it is the application.

The number of byte code instructions generated by the Java compiler as shown in Table 7 does not have any relation with the actual number of instructions executed. After profiling the Java application with Oracle JRockit Mission Control [17] we were able to determine that the function Sphere::SetPosition has been optimized by the runtime HotSpot JIT compiler but it was not clear what optimizations were performed. We also compiled the Java application into native code using the gcj [20] compiler version 4.4. gcj and g++ compilers use the same back-end, and the assembly code generated for the function Sphere::SetPosition by the gcj compiler was exactly the same as generated by the g++ compiler. Therefore in the next paragraph we discuss only, some of the differences in the generated assembly code, by the Fortran and the C++ compilers.

The number of instructions are more than double in the assembly code generated by the Fortran compiler than the assembly code generated by the C++ compiler as shown in Table 7. The results of profiling in Table 6 also shows that the number of instructions executed by the Fortran function Sphere::SetPosition are double than the C++ function Sphere::SetPosition. This gives one justification why Fortran is slower than C++ and Java. We can argue that the difference in number of instructions may not be important because almost all processors now a days have pipelines [21] and can execute instructions out-of-order [21].

- Carefully examining the instructions generated by the C++ compiler reveals that the function have been optimized. It uses SIMD registers (see note to Table 7) and the instructions are not dependent [3] on each other, and therefore can be pipelined [3].

- Whereas the instructions generated by the Fortran compiler use general registers and have multiple
Table 7: Generated Assembly (C++ and Fortran) and Byte Code (Java) for the Function `Sphere::SetPosition` Implemented as Part of the Simple Object Oriented Application Shown in Figure 7. Compilers used were `g++`, `gfortran` version 4.6 and Oracle `javac` version 1.6 on Linux 2.6.32 installed on Intel Core 2 Duo 64 bit machine. Optimization level O3 was used by the g++ and the gfortran compilers.

dependencies. The graph in Figure 8 shows these dependencies. Therefore these instructions cannot be efficiently scheduled \cite{3} for pipelining.

We will not go into the details why the Fortran compiler is not able to optimize this part of the code because it is out of the scope of this paper, but we point out some directions for future research to improve the object-oriented Fortran compilers. In object-oriented languages the selection of a target function in a dynamic dispatch \cite{28} is very important and is only known at runtime \cite{23, 19}. The compiler knows the abstract type of the object but not the concrete type. In our example the function `Shape::SetPosition` is abstract and is known to the compiler at compile time. The concrete type is the function `Sphere::SetPosition` where it is implemented and is only known at runtime. Therefore it can indirectly cause a compiler (which lacks the required analysis to get the information) to produce a poorly optimized code as in this case with the Fortran compiler.

Every modern compiler \cite{3} has a front-end and a back-end. The same back-end can be used for different languages that have different front-ends. Another argument that we make here is the use of the same back-end
by \texttt{g++} and \texttt{gfortran} compilers. Our profiling and generated assembly code analysis confirms that this is the case. Both the applications, in C++ and Fortran, are linked with the same libraries, except few, such as \texttt{cpu\_time}, \texttt{set\_args}, \texttt{set\_options} and \texttt{os\_error} etc. Most of the assembly code generated by both the compilers is similar but with some major differences. One of them is mentioned above and listed in Table \ref{table:profiling}. The other major difference is the number of instructions generated for the \textit{main} function. The \texttt{g++} compiler generated 88 instructions whereas the \texttt{gfortran} compiler generated 122 instructions for the \textit{main} function. This difference is also evident from the number of instructions executed in the \textit{main} function by C++ and Fortran applications as shown in Table \ref{table:instructions}.

The number of assembly instructions (listed in Appendix \ref{app:assembly}) generated for the following two very similar loops in the \textit{main} function are: by \texttt{gfortran} 42 and by \texttt{g++} 28. This loop is executed 393216000 times as shown in Table \ref{table:execution}. This explains why the number of instructions executed by the Fortran \textit{main} function are almost double than the number of instructions executed by the C++ \textit{main} function, as shown in Table \ref{table:instructions}.

\begin{verbatim}
Fortran:
call cpu\_time (time\_start)
do n1 = 1, NUM\_SPHERES
  COUNT = COUNT + 1
  n = n1
  call sphere\_t(n1)\%SetPosition(n,n+1.0,n+2.0)
end do
COUNT = COUNT / 100
call cpu\_time (time\_end)

C++:
clock\_t time\_start = clock();
for (int n1 = 0; n1 < NUM\_SPHERES; n1++) {
  COUNT = COUNT + 1;
  n = n1;
  sphere\_t[n1].SetPosition(n,n+1.0,n+2.0);
}
COUNT = COUNT / 100.0;
clock\_t time\_end = clock();
\end{verbatim}
This part of the code is exactly where the application is making a virtual function call \texttt{Sphere::SetPosition}. This analysis further confirms that Fortran compiler is not optimizing the virtual function call overheads. One explanation for this is: that the front-end of the Fortran compiler is not communicating enough information (or similar information as the C++ compiler) to the back-end, which is required by the compiler to optimize the generated code for object handling and processing.

Based on the discussion in the above paragraphs we provide some pointers, for further exploration, and list some of the techniques that can improve such a code, as follows: (1) A better interprocedural analysis \cite{3} such as complete information about the inheritance graph and the methods defined in each class. (2) Optimizations at link time, i.e: machine code optimization where more information is available about dynamic link libraries. (3) An improved front-end which collects and communicates enough information to the back-end for optimization. (4) Devirtualization \cite{27}. For further information on optimizations for object-oriented languages the reader is referred to \cite{27,4,1,23,22}.

Based on the discussion and the results presented in this Section and Section \cite{4} we reject the hypothesis \textbf{H1} and approve the hypothesis \textbf{H2} that \textit{there are some differences in the object-oriented features of Fortran from C++ and Java that does impact it’s performance and hence there is a need to improve it further to make it equivalent to C++ and Java.}

We have provided an insight into the simple object-oriented application implemented in the three languages in the hope that this will motivate language implementers and compiler developers to improve Fortran object handling and processing, and hence make it's use more prolific and general.

References


**Appendices**

A Generated assembly code of part of the simple object-oriented application listed in Figure 7 by g++ and gfortran compilers

**g++:**

```plaintext
1 400880: e8 6b ff ff ff
2 400885: 49 89 df
3 400888: 49 89 c4
4 40088b: 45 31 f6
5 40088e: 66 90
6 400890: 00
7 400891: 66 90
8 400892: f2 0f 10 05
9 400893: a8 03 00
10 400896: movsd 0x3a8(% rip),%xmm0
11 400898: mov (%rip),%rax
12 400899: movsd 0x3a5(% rip),%xmm2
```

```plaintext
13 mov (%rip),%rax
14 callq 4007f0 <clock@plt>
```
gfortran:

```
1 4009bd: 00 e8 2e fe ff ff callq 4007f0 <__gfortran_cpu_time@plt>
2 4009c2: 66 0f 1f 44 00 00  movw 0x0(%rax,%rax,1)
3 4009c8: f2 0f 10 05 10 03 00 movsd 0x310(%rip),%xmm0
4 4009d0: 41 8d 45 01 lea 0x1(%r13),%eax
5 4009d4: f2 0f 10 0d 04 03 00 movsd 0x304(%rip),%xmm1
6 4009d8: 48 8d 8c 24 08 02 00 lea 0x208(%rip),%rcx
7 4009de: 00
8 4009e4: f2 0f 58 04 24 addsd (%rsp),%xmm0
9 4009e9: 48 8d 94 24 00 02 00 lea 0x200(%rip),%rdx
10 4009f0: 00
11 4009f9: 48 8d be 24 f0 01 00 lea 0x1f0(%rip),%rdi
12 400a01: 00
13 400a09: be 18 21 60 00 mov $0x602118,%esi
14 400a0f: 48 c7 84 24 f8 01 00 movq 0x6020c0,0x1f8(%rip)
15 400a15: 00 00 00 60 00
16 400a1a: f2 0f 11 04 24 movsd %xmm0(%rip)
17 400a1f: cvttsi2sd %eax,%xmm0
18 400a20: 49 8d 44 6d 00 lea 0x0(%r13,%r13,2),%rax
19 400a25: 49 83 c5 01 add 0x1,%r13
20 400a2a: 48 8d 04 c3 lea (%rbx,%rax,8),%rax
21 400a2f: 48 89 84 24 f0 01 00 mov %rax,0x1f0(%rip)
22 400a34: 00
23 400a39: f2 0f 58 c8 addsd %xmm0,%xmm1
24 400a3e: f2 0f 11 05 e4 16 20 movsd 0x2ac(%rip),%xmm0
25 400a43: 00
26 400a48: f2 0f 58 05 ac 02 00 movsd 0x2ac(%rip),%xmm0
27 400a4d: 00
28 400a52: f2 0f 11 8c 24 00 02 movsd 0x2ac(%rip),%xmm0
29 400a57: 00
30 400a5c: f2 0f 11 84 24 08 02 movsd 0x2ac(%rip),%xmm0
31 400a61: 00 00 00
32 400a66: e8 5d fe ff ff callq 400950 <__sphere_setposition,sphere>
33 400a6b: 49 81 fd 04 01 00 00 cmp 0x1f4,%r13
34 400a70: 48 89 84 24 f0 01 00 mov %rax,0x1f0(%rip)
35 400a75: 00
36 400a7a: 00
37 400a85: 31 c0 xor %eax,%eax
38 400a8a: 48 8d 75 00 02 00 movsd 0x279(%rip),%xmm0
39 400a90: 00
40 400a95: f2 0f 11 04 24 movsd 0x2ac(%rip),%xmm0
41 400a9a: e8 6f fd ff ff callq 4007f0 <__gfortran_cpu_time@plt>
```