High-Performance Java Codes for Computational Fluid Dynamics [*]

Christopher J. Riley
Blue Ridge Numerics, Inc.
Charlottesville, VA 22901
chris@cfdesign.com

Siddhartha Chatterjee
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
sc@us.ibm.com

Rupak Biswas
NAS Systems Division, MS T27A-1
NASA Ames Research Center
Moffett Field, CA 94035
rbiswas@nas.nasa.gov

ABSTRACT

The computational science community is reluctant to write large-scale computationally-intensive applications in Java due to concerns over Java's poor performance, despite the claimed software engineering advantages of its object-oriented features. Naive Java implementations of numerical algorithms can perform poorly compared to corresponding Fortran or C implementations. To achieve high performance, Java applications must be designed with good performance as a primary goal. This paper presents the object-oriented design and implementation of two real-world applications from the field of Computational Fluid Dynamics (CFD): a finite-volume fluid flow solver (LAURA, from NASA Langley Research Center), and an unstructured mesh adaptation algorithm (2D_TAG, from NASA Ames Research Center). This work builds on our previous experience with the design of high-performance numerical libraries in Java. We examine the performance of the applications using the currently available Java infrastructure and show that the Java version of the flow solver LAURA performs almost within a factor of 2 of the original procedural version. Our Java version of the mesh adaptation algorithm 2D_TAG performs within a factor of 1.5 of its original procedural version on certain platforms. Our results demonstrate that object-oriented software design principles are not necessarily inimical to high performance.

1. INTRODUCTION

The Java programming language has many features that are attractive for both general-purpose and scientific computing. Among them are: support for object-oriented concepts such as inheritance, encapsulation, and polymorphism that allow the abstraction of common physical concepts and the development of reusable class libraries for them; the architecture-neutrality of Java byte code, which enables portability across multiple platforms; garbage collection, which simplifies memory management; and language-level support for multithreading, which allows parallel applications to be developed more easily in Java. However, scientific programs written in Java usually run slower than corresponding programs written in Fortran and C, due either to the intrinsic overhead of these features or to the relative immaturity of current Java Virtual Machine (JVM) implementations. Because high performance is a primary concern in computational science, that community has been reluctant to adopt Java as the language of choice for numerical applications.

To demonstrate the viability of high-performance computing in Java and to encourage its greater adoption in the computational science community, several authors have ported numerical libraries to Java [1, 5, 19], written Fortran-to-Java translators [8, 10], developed compilation technology for Java [6, 7, 25], and written class libraries to address deficiencies in the Java language for numerical computing [26]. Although these studies demonstrate the potential of Java for high-performance computing, several factors limit their usefulness in determining whether large-scale scientific applications written in Java can achieve high performance. First, numerical and class libraries form only part of such applications. The only previous ports of large-scale codes to Java that we know of are a geophysical simulation by Jacob et al. [20] and a parallel multi-pass renderer by Yamauchi et al. [31]. Second, "line-by-line" translations of procedural codes written in Fortran or C to Java do not exploit object-oriented techniques, one of the main advantages to programming in Java.

The primary goal of the work described in this paper is to demonstrate that realistic scientific applications can be written in Java that make full use of the language's object-oriented capabilities and still show good performance on current standards-conforming JVM implementations. Secondary goals include characterizing the performance and identifying the bottlenecks in different JVM implementations, identifying design principles for portable high-performance object-oriented software in Java, and making available additional benchmarks for the high-performance Java community. It is our thesis that in order to achieve high performance with such applications, one must consciously design for performance in both the definition and the implementation of the software components that comprise them. We choose two real-world examples from the field of Computational Fluid Dynamics (CFD) for our study. The first example is the Langley Aerothermodynamic Upwind Relaxation Algorithm (LAURA) [9, 12], a finite-volume flow solver developed at NASA Langley Research Center for multiblock, structured grids. LAURA has been widely used to compute hypersonic, viscous, reacting-gas flows over reentry vehicles such as the Shuttle Orbiter [14], the Mars Pathfinder [24], and the X-33 Reusable Launch Vehicle [15]. The second example is the Two-Dimensional Triangular Adaptive Grid (2D_TAG) code [27] developed at NASA

[*]This work was supported in part by NSF Grant CCR–9711438 while the first two authors were at the University of North Carolina at Chapel Hill.

Java 2001 Palo Alto, CA USA
Copyright 2001 ACM 0-89791-88-6/97/05 ...$5.00.

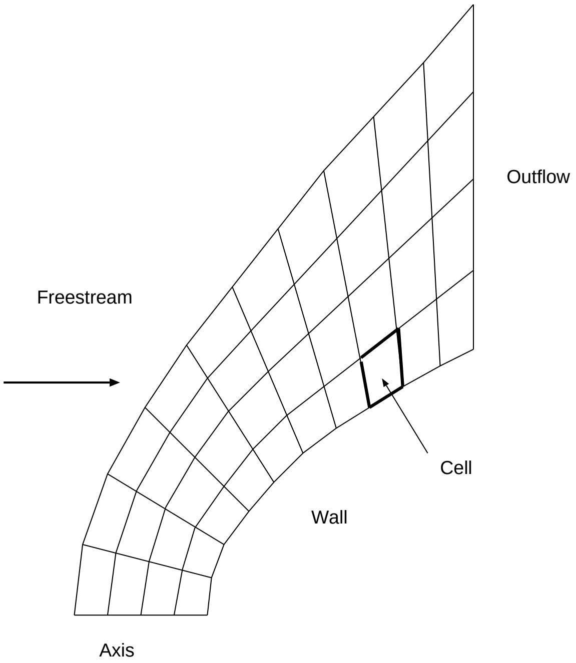

Figure 1: LAURA grid over a parabola (2-D).

Ames Research Center for adaptation of unstructured meshes. This code is a simplified version of a three-dimensional mesh adaption algorithm [4] developed primarily for helicopter flowfield simulations. Both LAURA and 2D_TAG are currently used in government and industry and are well-documented.

The remainder of the paper is organized as follows. Sections 2 and 3 describe the algorithms and data structures of the original versions of LAURA and 2D_TAG and their object-oriented Java implementations. Section 4 presents performance measurements of the two codes on several platforms. Section 5 concludes and presents plans for future work.

2. LAURA

LAURA [9, 12] is a finite-volume, shock-capturing algorithm for the steady-state solution of inviscid or viscous, hypersonic flows on rectangularly-ordered, structured grids. It is written in Fortran, and parallel implementations exist for vector architectures such as the Cray C90 [13] and for machines distributed across a network using MPI [28]. Gas chemistry options include perfect gas, equilibrium air, and air in chemical and thermal nonequilibrium.

2.1 Algorithm

LAURA solves the system of partial differential equations governing fluid flow numerically by first discretizing the solution domain using a collection of *blocks*, *points*, and *cells* as shown in Figure 1. A block is defined as a rectangular collection of cells. Figure 1 shows a single block. Because the grid is structured, a multidimensional array is the natural data structure to use for storage. Solution variable unknowns are assumed to lie at the center of a cell. Numerically approximating the physically relevant fluxes and stresses acting on the cell walls leads to a finite-volume formulation of the governing conservation laws. In LAURA, the upwind-biased inviscid flux is constructed using Roe's flux-difference splitting [29] and Harten's entropy fix [16] with second-order corrections based on Yee's symmetric total-variation-diminishing (TVD) scheme [32].

To drive the residual of the finite-volume approximation scheme to zero simultaneously at all the cells of the domain, LAURA uses a point-implicit relaxation strategy obtained by treating the variables at the local cell center L at the advanced iteration level and using the latest available data from neighboring cells. This governing relaxation equation is

$$\boldsymbol{M}_L \, \delta\boldsymbol{q}_L = \boldsymbol{r}_L \tag{1}$$

where $\boldsymbol{M}_L$ is the $n \times n$ point-implicit Jacobian, $\boldsymbol{q}_L$ is the vector of conserved variables, $\boldsymbol{r}_L$ is the the residual vector, and n is the number of unknown variables. The value of n varies from 5 for perfect gas flows to 15 for chemically-reacting flows with eleven species.

The point-implicit strategy has several implications compared with other solvers (e.g., tridiagonal, conjugate-gradient). First, the memory requirements are lower, which allows solutions using nonequilibrium chemistry to be obtained over larger grids. This savings in memory is traded off against an increase in the number of iterations (albeit faster iterations) required to reach convergence. Second, iterations across blocks do not have to be synchronized. This allows for an efficient parallel implementation.

The algorithm may be summarized as follows:

1. Sum the fluxes for each cell in the domain to get the residual $\boldsymbol{r}_L$.

2. Compute the change in conserved variables at each cell, $\delta\boldsymbol{q}_L$, by solving equation (1) using Gaussian elimination.

3. Update the vector of conserved variables $\boldsymbol{q}_L$ at each cell.

These steps constitute an iteration in which the solution variables at all cells in the domain are integrated to the next time level.

While the above description captures the basic steps in the algorithm, the original Fortran version of LAURA contains additional gather and scatter steps at each iteration designed to take advantage of vector architectures. At each iteration, two-dimensional slabs of size *IMAX* × *JMAX* are copied into one-dimensional temporary arrays, the fluxes and variables of each slab are updated, and then the temporary values are put back into its original two-dimensional slab. This approach works very well on vector architectures as it creates large vectors on which the vector processors can operate. However, the extraneous copying hurts performance on current, RISC-based architectures that use caches. To make a more accurate measure of the relative performance of the Java version to the original procedural version, we wrote a corresponding C version of LAURA that eliminates the unnecessary copying.

2.2 Java Version

The object-oriented design of the Java version of LAURA begins with recognizing that the various geometric abstractions used (e.g., block, cell, cell face, and point) are candidates for encapsulation. Thus, we model the grid with *Block*, *Cell*, *CellFace*, and *Point* classes. We use the multidimensional array class library developed at IBM [26] to manage the three-dimensional arrays of geometric objects contained in a block. We chose not to use the native Java arrays of arrays because of their inefficiency. In addition, the array class library from IBM contains methods that allow portions of the array to be accessed. These methods are convenient for accessing cells and points on a boundary face, for instance.

Another facet of object-oriented design is determining which components of the system are likely to change and isolating the effects of their changes from the rest of the system. Because LAURA

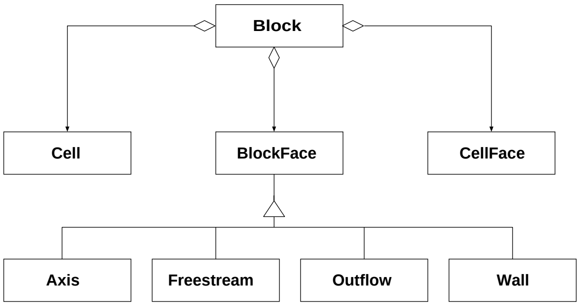

Figure 2: Block class abstraction.

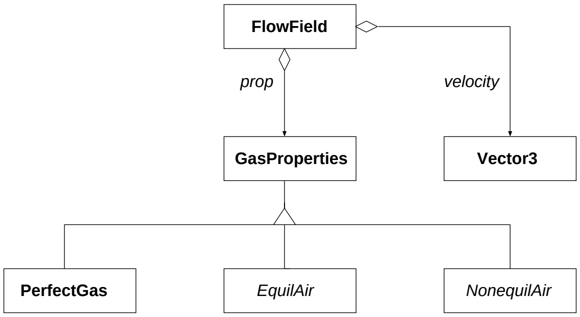

Figure 3: Flowfield class abstraction.

supports multiple gas chemistry options as well as multiple boundary conditions, we wrote two abstract classes, *GasProperties* and *BlockFace*, to minimize the effects of adding new chemistry models and boundary conditions to the software. The *GasProperties* class is subclassed for each particular chemistry model. Currently, only the perfect gas model is supported in the Java version as it is the only gas model available in the version of LAURA released to us by NASA Langley Research Center. The *BlockFace* class is subclassed for each of four different boundary conditions. Diagrams showing the geometry and flowfield class abstractions are given in Figures 2 and 3. The *EquilAir* and *NonequilAir* classes pictured in Figure 3 are not currently implemented, but illustrate how subclassing would be used to extend the abstract *GasProperties* class in a full-featured version of LAURA. Where possible, only references to the abstract base classes are made.

2.2.1 Design Patterns

Gamma et al. [11] define *design patterns* as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context." Our Java code uses the *Factory* creational pattern to manage the construction of the various geometric objects and the *Singleton* creational pattern to control access to atmospheric constants. However, there are additional design patterns that we did not use. The *Iterators* pattern would be useful in sweeping over a block, but the multidimensional array package we use does not contain them. The *Strategy* pattern would be useful in managing changes to the choice of relaxation algorithms. We did not use it in our version because the original Fortran version of LAURA supports only one relaxation algorithm.

2.2.2 Multithreading

We added multithreading by subdividing a single block into multiple blocks and assigning a Java thread to each sub-block. This follows the domain decomposition strategy used to parallelize the original Fortran version of LAURA for distributed machines [28]. In this multithreaded version, two *Block* objects will share a single *BlockFace* object. Conflicts may occur if each *Block* tries to modify the shared *BlockFace* at the same time. Java provides a monitor locking mechanism with the `synchronized` keyword for performing mutual exclusion on an object. Use of `synchronized` for the appropriate methods is the safest approach to handling this block interface conflict in that it prevents an inconsistent update of the boundary cells and cell faces at the boundary between two blocks. However, by using the point-implicit relaxation strategy, blocks may be relaxed asynchronously. Blocks do not need to be kept in lock step at the same iteration level. For this algorithmic reason, our code does not use the `synchronized` keyword.

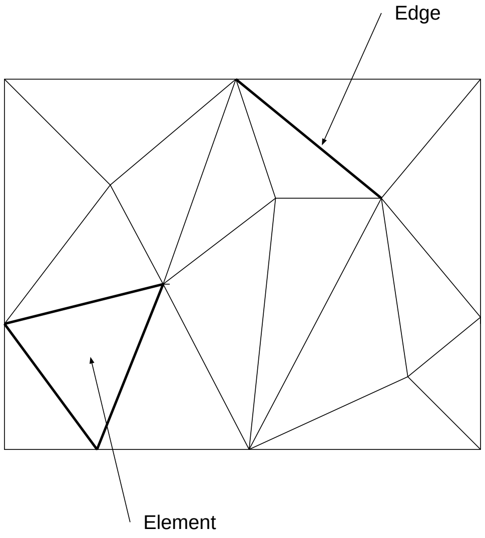

Figure 4: An example of an unstructured mesh.

3. 2D_TAG

2D_TAG [4, 27] is a two-dimensional, unstructured mesh adaptation scheme that locally refines and/or coarsens the computational grid. It is written in C and is a simplified version of a corresponding three-dimensional algorithm. Biswas and Strawn [4] describe the details of the algorithm. Oliker and Biswas [27] discuss its performance under different parallel processing strategies.

It is important to note here the different roles of a mesh adaptation algorithm like 2D_TAG and a flow solver like LAURA. A mesh adaptation code is a tool used to support the flow solver by concentrating the solver's computation in regions of interest such as shocks and shear flows. During the computation of a steady-state flow field, the mesh is adapted infrequently, on the order of once every several hundred solver iterations. In this scenario, the mesh adaptation may consume 5% of the total running time. For time-dependent calculations, the mesh is adapted much more frequently and may account for 30–40% of the total computation time. Good performance of both the solver and of the mesh adaptation scheme is therefore desirable.

3.1 Algorithm

A mesh is described as a collection of two-dimensional triangular elements, as shown in Figure 4. The mesh is *unstructured* in that there is no logical ordering of an element and its neighbors as with the structured grids used with LAURA. The mesh is *locally adapted*. This involves adding points to the existing grid in regions

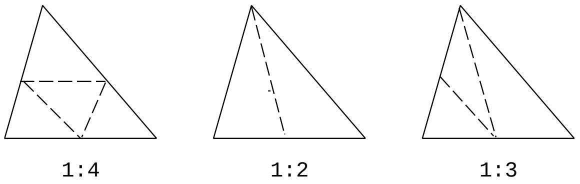

Figure 5: Triangular refinement.

where some user-specified error indicator is high, and removing points from regions where the indicator is low. The advantage of this strategy is that relatively few mesh points need to be added or deleted at each refinement/coarsening step. A disadvantage is that complicated data structures and logic are required to keep track of the mesh objects (e.g., vertices, edges, elements) and their neighbors as the objects are added and removed. The algorithm involves much "pointer chasing" leading to irregular and dynamic data access patterns.

Triangular elements may be refined by bisecting one or more of its edges. An *isotropic* subdivision bisects all three of an element's edges to form four congruent "child" triangles. An *anisotropic* subdivision bisects one or two of the edges to form two or three child triangles. Figure 5 depicts this process of triangle refinement. Anisotropic subdivisions are required as a transition from isotropically-refined elements to unrefined elements. Coarsening is the process of deleting child edges and elements and restoring their parents.

The elements and edges of the mesh are stored as forests, with a tree for subsets of elements and edges. Using trees simplifies the coarsening process. Roots of trees in the forest correspond to the original unrefined elements and edges. Each level of a tree in the forest corresponds to a level of refinement. Leaves of the trees represent elements and edges at the maximum level of refinement. Leaf elements and edges are the only ones that may be refined or coarsened. Vertices do not have parents or children and so are stored in an array or list.

The refinement algorithm may be summarized as follows:

1. For each leaf edge of the mesh that meets a specified error criterion, mark edge for refinement.

2. Subdivide elements based on their edges that are marked for refinement.

3. Add new vertices, edges, and elements to the mesh.

The coarsening algorithm is similar. As described, this mesh adaptation algorithm involves multiple passes over the lists of edges and elements.

3.2 Java Version

As in the case of LAURA, the design of the object-oriented Java version of 2D_TAG begins with modeling the geometry. *Mesh*, *Vertex*, *Edge*, and *Element* classes model the different mesh objects. Initially, we used the various *List* implementations from the Java Collections package [30] to manage the lists of mesh objects and their neighbors. After seeing that the initial performance of this version was poor, we realized that we did not need a general, resizable list in most situations (e.g., there are always three edges in a triangle). As a result, we chose to use native Java arrays or local variables wherever possible.

Each *Element* and *Edge* object contains references to its child objects as well as to its parent. The various refinement and coarsening error tests are defined as subclasses of a *Predicate* class. To simplify the traversals over the edges and elements of the mesh during the refinement and coarsening steps, we initially decided to use specialized iterator classes or filters defined to return only those objects that satisfy a *Predicate* (e.g., element is a leaf, edge meets error criteria). As with the use of general lists, we recognized that iterators are not necessary in our case and chose not to use them.

Property	*Value*
Velocity	3000 *m/s*
Density	0.001 kg/m^3
Temperature	250 *K*
Wall Temperature	500 *K*

Table 1: Freestream conditions for LAURA test case.

3.2.1 Multithreading

We tried two approaches in developing a multithreaded version of 2D_TAG. The first strategy assigns groups of mesh objects to individual threads and handles conflicts between shared edges and vertices by making *Edge* and *Vertex* methods `synchronized`. This puts locks on the shared objects of the mesh and mimics the NO_COLOR strategy described by Oliker and Biswas [27]. The second strategy first statically partitions the mesh into submeshes using the mesh partitioning tool METIS [22]. Only the edges and vertices at partition boundaries require locking. This reduces the amount of synchronization considerably. For instance, only 1% of the edges and 6% of the vertices of our test mesh are shared when the mesh is split into four partitions.

4. PERFORMANCE

We now compare and characterize the sequential and parallel performance of the object-oriented Java versions of LAURA and 2D_TAG with their original procedural versions written in Fortran and C. Note that we have both a Fortran and a C version of LAURA, as mentioned in Section 2.1. We use the C version as the benchmark for our performance comparisons, because it does not contain the unnecessary copying of data present in the Fortran version. Execution times from the Fortran version are included for completeness. In this section, we describe the test cases, present the performance results for several architectures, and discuss the reasons for both good and poor performance.

4.1 Test Cases

We use the computation of three-dimensional, viscous, perfect gas flow over a paraboloid at 10 degrees angle of attack as a test case for LAURA. The initial flow field is set to the freestream conditions listed in Table 1 and 200 iterations are computed using first-order accuracy. A variety of grid sizes are used ranging from 1000 cells (10×10×10) to 64,000 cells (40×40×40). Figure 6 shows a view of the 10×10×10 grid's upper and lower symmetry planes. The largest of these grids, which contains only 64,000 cells, is still relatively small compared to CFD solutions that require millions of cells. However, these grids can be thought of as constituting a single block of a multiblock solution and are a reasonable size for computations on a workstation or PC. We also tested LAURA using both single precision (32-bit) and double precision (64-bit) floating point arithmetic as both versions are used in actual flow simulations. Single precision is used for inviscid flow computations where the mesh is not highly refined, while double precision is typically used for viscous or reacting gas flows. As there was little difference in the running times of the two versions, we present

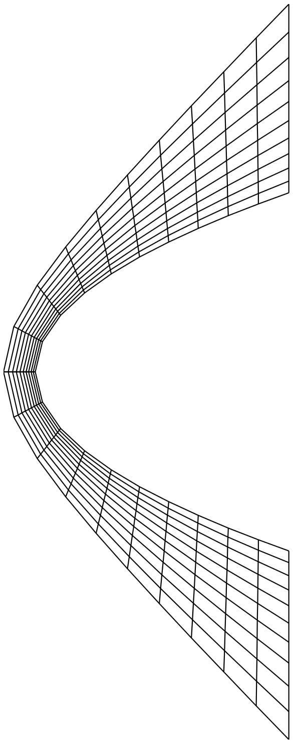

Figure 6: Side view of the 10×10×10 grid around the paraboloid.

Mesh	*Vertices*	*Triangles*	*Edges*
Initial	14,605	28,404	43,009
Level 1	26,189	59,000	88,991
Level 2	62,926	156,498	235,331
Level 3	169,933	441,147	662,344
Level 4	380,877	1,003,313	1,505,024
Level 5	488,574	1,291,834	1,935,619

Table 2: Progression of grid sizes through five levels of adaptation.

only the single precision results.

We use the computational mesh over an airfoil to test the performance of 2D_TAG. This is the same test case used by Oliker and Biswas [27]. For an actual flow simulation over an airfoil traveling at transonic Mach numbers, shocks form on both the upper and lower surfaces of the airfoil. The mesh is typically refined in the area containing the shocks as well as around the stagnation point located at the leading edge of the airfoil. This scenario is simulated by geometrically refining the mesh in these regions. The actual test case consists of reading the initial coarse mesh into 2D_TAG and proceeding through five levels of refinement. Table 2 gives the resulting grid sizes at each level of refinement. Note that the final mesh is more than 40 times larger than the initial one. Figure 7 shows a close-up view of the initial mesh.

4.2 Testing Environment

Table 3 lists the platforms and JVMs we used for measuring performance. All JVMs were run using Just-In-Time (JIT) compilation and with garbage collection enabled. All machines were relatively unloaded at the time of each test, and several runs were made at each test condition with the best time recorded. The initial and maximum heap sizes were set using the `-Xms` and `-Xmx` options to values on the order of 500 MB to discourage excessive garbage collection. The typical variation in run times was less than 1 second for run times on the order of 60 seconds. All JVMs are freely available except for Jalapeño [2, 7], a research JVM developed at IBM T.J. Watson Research Center. Although the Jalapeño JVM contains an optimizing JIT compiler, it is not tuned for scientific codes. For instance, it does not perform traditional optimizations for scientific computing such as loop unrolling. Jalapeño also has an adaptive compiler using dynamic feedback[3] that may recompile methods at different optimization levels during the course of a code's execution. We chose to use Jalapeño's optimizing compiler instead, because it provided us with more control over optimization options.

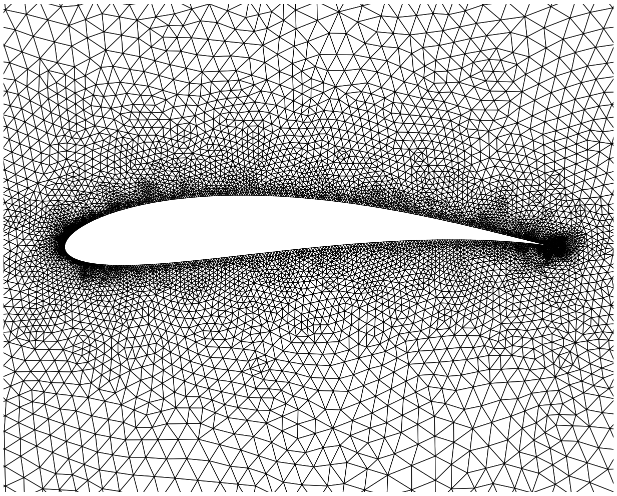

Figure 7: Close-up view of the initial mesh around the airfoil.

4.3 LAURA Results

Tables 4 and 5 give the performance of the single-precision, native and Java versions of LAURA on the different platforms. Running times are normalized by the total number of cells and iterations. The times are normalized to help show variations in running time with grid size and to help determine running time for different grid sizes and iteration counts. Recall that the C version of LAURA (see Section 2.1) does not contain the copying to and from temporary work arrays that is present in the original Fortran version. We consider the performance of the C version to be the benchmark against which the performance of the Java version should be compared.

Several observations regarding the timing measurements are in order.

1. As shown in Table 5, the performance of the Java version of LAURA is within a factor of 3 of the corresponding C version on the Sun Ultra and the Pentium when using the Sun JVM and within a factor of 2.5 on the Pentium when using the IBM JVM. The fastest Java version on the PowerPC with the Jalapeño JVM is still 3 times slower than the native C version. The focus of the Jalapeño JVM research project has not been on optimization of scientific codes and floating point calculations.

	Sun Ultra	*SGI*	*Pentium*	*PowerPC*
Processor	Sparc	R12000	Pentium III	PowerPC RS 64 II
Processor Speed	300 MHz	300 MHz	700 MHz	262 MHz
Number of CPUs	2	32	4	12
Memory	256 MB	16 GB	1 GB	8 GB
Operating System	SunOS 5.7	IRIX 6.5	Red Hat Linux 6.2	AIX
JVM	Sun JDK 1.3.0-beta Sun JDK 1.2.1	SGI SDK 1.2.2	Sun JDK 1.3.0 IBM JDK 1.3	Jalapeño [2, 7]
Fortran/C Compiler	f77/cc Sun Workshop 5.0	f77/cc MIPSpro 7.30	g77/gcc GCC 2.96	xlc
Fortran/C Compiler Switches	-fast	-Ofast	-O3	-O4

Table 3: Machines used to test LAURA and 2D_TAG.

Machine	*JVM*	*Grid*	Time (μsec)	Ratio to C version
Sun Ultra	Sun JDK 1.3.0-beta	10×10×10	215.6	3.51
		20×20×20	183.0	2.86
		40×40×40	182.8	2.63
	Sun JDK 1.2.2	10×10×10	185.2	3.01
		20×20×20	181.8	2.84
		40×40×40	183.6	2.65
SGI	SGI SDK 1.2.2	10×10×10	141.6	9.51
		20×20×20	151.3	10.02
		40×40×40	172.9	10.74
Pentium	Sun JDK 1.3.0	10×10×10	49.3	2.58
		20×20×20	55.8	2.83
		40×40×40	60.0	2.78
	IBM JDK 1.3.0	10×10×10	39.4	2.06
		20×20×20	39.8	2.02
		40×40×40	44.0	2.04
PowerPC	`-O0`	32×32×32	201.1	4.24
	`-O1`		154.0	3.25
	`-O2`		161.9	3.42
	`-O2 -no_bounds_check`		155.4	3.28
	`-O2 -no_bounds_check -allow_fma`		143.9	3.04
	`-O1 -no_bounds_check -allow_fma`		138.2	2.92

Table 5: Time per cell per iteration (μsec) of Java version of LAURA on test platforms. Smaller numbers indicate higher performance.

Machine	*Grid*	Time (μsec)	
		Fortran	C
Sun Ultra	10×10×10	69.8	61.5
	20×20×20	86.6	64.1
	40×40×40	89.8	69.4
SGI	10×10×10	25.7	14.9
	20×20×20	25.2	15.1
	40×40×40	29.5	16.1
Pentium	10×10×10	30.0	19.1
	20×20×20	31.3	19.7
	40×40×40	44.1	21.6
PowerPC	32×32×32	N/A	47.4

Table 4: Time per cell per iteration (μsec) of native, single-precision versions of LAURA on test platforms. Smaller numbers indicate higher performance.

2. The relative performance of the Java version on the SGI is poor. Several factors may account for this. One is the obvious reason that the implementation of the JVM on the SGI may be poor. The other reason is that the Fortran and C compilers on the SGI may be much better than the JIT compiler used by the JVM and better able to take advantage of the underlying architecture.

3. The time spent copying data to and from temporary arrays in the original Fortran version is significant, as seen in the difference between the performance of the C and Fortran versions in Table 4.

4. Two desired changes to the Java language that are frequently mentioned by the scientific community are the removal of array bounds checks and the ability to used fused multiply-add (FMA) instructions [21]. With the Jalapeño JVM [2, 7] on the PowerPC, we can explore the effects of these proposed language changes on the performance of LAURA. Comparing the results with `-allow_fma` turned on and off shows that allowing FMA instructions reduces the running time by

JVM	*Grid*	IBM [26]	Colt [18]	Native Java
Sun JDK 1.3.0	10×10×10	49.3	51.6	51.7
	20×20×20	55.8	57.5	58.0
	40×40×40	60.0	64.9	64.7
IBM JDK 1.3.0	10×10×10	39.4	39.4	39.5
	20×20×20	39.8	40.0	40.2
	40×40×40	44.0	45.8	46.2

Table 6: Time per cell per iteration (μ*sec*) of LAURA-Java using different array packages on Pentium.

7%. Determining the effect of removing array bounds checks is more difficult in that many of the array bounds checks may have already been removed by the `-O1` and `-O2` optimization levels. Because a run with the `-O0 -no-bounds-checks` options was not made, we hesitate to draw definitive conclusions from the data except to say that array bounds checks (difference between `-O2` and `-O2 -no-bounds-checks` times) account for at least 4% of the total running time.

4.3.1 Array Packages

We next examine the effect of the multidimensional array package on the performance of LAURA. Table 6 lists the performance of several versions of LAURA on the Pentium, each using a different array package to manage the collections of geometric objects found in a block. The performance of LAURA using native Java arrays is also listed. The conventional wisdom is that native Java arrays are inefficient due to the array bounds checking that Java requires and the possible lack of data locality of the Java "array of arrays" [26]. While this statement is certainly true, it must be evaluated in terms of the actual application. Unlike many other CFD applications, we are not using multidimensional arrays in LAURA to store primitive floating-point values such as pressure, velocity, and density, nor are we using arrays to perform matrix factorization or multiplication. In the case of LAURA, we are using multidimensional arrays as containers of *Cell*, *CellFace*, and *Point* objects which in turn store the primitive data. Most of the running time is spent in getting this data from memory to the registers and then executing floating point operations with them. The time spent in `get` and `set` operations in the various array packages is small in comparison. Therefore, the choice of array packages has little impact on the performance of LAURA.

4.3.2 Multithreaded Performance

Figure 8 shows the multiprocessor speedup of the Java version of LAURA on the 4-processor Pentium machine and the 12-processor PowerPC machine. As mentioned in Section 2.2.2, our Java version of LAURA does not use `synchronized` methods due to the asynchronous relaxation allowed by the point-implicit relaxation strategy. With only minimal modifications to the code, we are able to achieve near ideal speedup using the Jalapeño JVM on the PowerPC and a modest speedup of around 3 on the 4-processor Pentium machine.

4.4 2D_TAG Results

Tables 7 and 8 list the performance of the C and Java versions of 2D_TAG on the various architectures. The "Startup" time represents the time spent reading in the original mesh and allocating and initializing relevant data structures. The "Adapt" time represents the time spent in refining the mesh. The total running time of a version is the sum of these times. The "Adapt Time Ratio" for a Java version is the ratio of its adaptation time to the adaptation time of the original C version. Smaller values of this ratio indicate better performance of the Java code. The "GC" time represents the time spent in garbage collection by the Java versions, as indicated by the `-verbosegc` flag. Due to memory constraints, the initial mesh is refined only three times on the Sun Ultra instead of five. We tested three Java versions: a procedural version obtained by a "line-by-line" translation of the original C version; an object-oriented (O-O) version as described in Section 3.2 that uses Java arrays to store *Element*, *Edge*, and *Vertex* objects; and an O-O version that uses linked lists to store the collections of mesh objects. Using linked lists avoids array bounds checks and also avoids the need to preallocate the storage for the final refined mesh. Because the original C version as well as the other Java versions use arrays for storage, they require the final mesh size to be specified at run time. The difference in performance between the three Java versions gives an imperfect indication of the performance "cost" of writing object-oriented code.

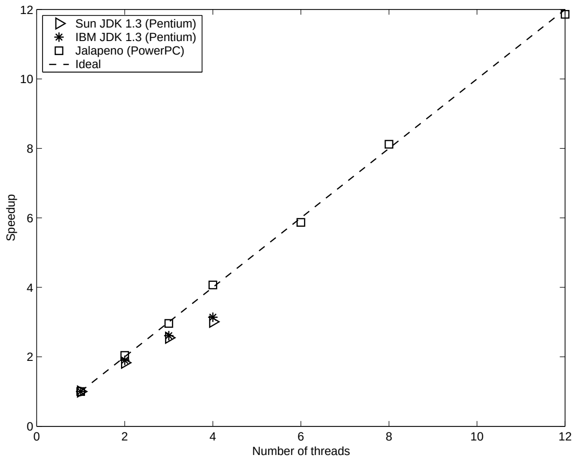

Figure 8: Speedup of LAURA-Java on 4-processor Pentium and on 12-processor PowerPC with Jalapeno. The grids used are 24×24×24 (Pentium) and 32×32×32 (PowerPC).

Machine	Startup (*sec*)	Adapt (*sec*)
Sun Ultra*	1.60	4.01
SGI	0.77	7.89
Pentium	0.58	3.75
PowerPC	1.94	7.17

Table 7: Running times of C version of 2D_TAG for five levels of refinement. Note that the grid is refined only three times on the Sun Ultra due to memory constraints.

Generally, the Java versions perform quite well, with the O-O versions running within 6% of the performance of native code on the PowerPC using the Jalapeño JVM and within 33% on the Pentium with the IBM JVM. In fact, the O-O versions actually outperform the Java procedural version on the PowerPC, SGI, and Sun Ultra. Examining the hardware counters on the SGI using `perfex` shows an increased number of Translation Lookaside Buffer (TLB) misses in the procedural version. We conjecture that the O-O version does a better job of achieving locality in the TLB by encapsulating data in the *Vertex*, *Edge*, and *Element* classes.

Machine	*JVM*	*Version*	Startup	Adapt	GC	Adapt Time
			(*sec*)	(*sec*)	(*sec*)	Ratio
Sun Ultra*	Sun JDK 1.3.0-beta	Procedural	2.26	9.68	0.80	2.41
		O-O (Array)	0.87	8.86	5.29	2.21
		O-O (List)	1.03	10.49	6.99	2.62
SGI	SGI SDK 1.2.2	Procedural	2.21	24.42	0.00	3.10
		O-O (Array)	0.96	15.12	0.00	1.92
		O-O (List)	0.97	13.80	0.00	1.75
Pentium	Sun JDK 1.3.0	Procedural	1.07	5.95	0.18	1.59
		O-O (Array)	0.52	17.66	12.50	4.71
		O-O (List)	0.61	19.22	14.34	5.13
	IBM JDK 1.3.0	Procedural	1.26	4.44	0.00	1.18
		O-O (Array)	0.41	5.09	0.00	1.36
		O-O (List)	0.43	4.99	0.00	1.33
PowerPC	`-O2`	Procedural	6.11	12.37	0.00	1.73
	`-O2 -no_bounds_check`		5.26	9.97	0.00	1.39
	`-O1`	O-O (Array)	0.80	7.87	0.00	1.10
	`-O1 -no_bounds_check`		0.79	7.82	0.00	1.09
	`-O1`	O-O (List)	0.87	7.63	0.00	1.06
	`-O1 -no_bounds_check`		0.85	7.60	0.00	1.06

Table 8: Running times of Java versions of 2D_TAG for five levels of refinement. Note that the grid is refined only three times on the Sun Ultra due to memory constraints.

One of the main differences between the procedural and O-O versions is the number and granularity of objects. The procedural version lies at one extreme in that there is only one object, namely, a *Mesh* object that encapsulates all of the original mesh adaptation algorithm and its data structures. The O-O version, on the other hand, is composed of a *Mesh* object containing collections of individual *Vertex*, *Edge*, and *Element* objects. Referring to Table 2, we observe that refining the test mesh five times results in the creation of almost 4 million objects. The original C version pre-allocates the required amount of storage by using a user-supplied overallocation factor that specifies the expected size of the final refined mesh relative to the original mesh, and ceases operation should this factor be found to be too small. The Java procedural version mimics this behavior, allocating multiple such arrays of primitive (i.e., non-object) types that are not subject to garbage collection. The O-O version uses this factor to pre-allocate an array of object references, but allocates the actual objects as needed. As the test case performs only mesh refinement, none of the objects created are actually destroyed during program execution.

Looking at the garbage collection (GC) times for the Sun JVM in Table 8 shows the impact of this large number of small objects on performance. The GC time accounts for over half of the execution time of the O-O version while the GC time used by the procedural version is negligible. Note that GC is largely unproductive in freeing heap storage, since no objects are ever destroyed. If the GC time could be avoided (as was possible in earlier JVM versions with the `-noasyncgc` command-line switch), the actual time spent refining the mesh would be quite similar between the procedural and O-O versions. Interestingly enough, no time is spent in GC when using the other JVMs (as reported by using the `-verbosegc` flag).

The startup time for the O-O version is actually smaller than the startup time for the original C version. This relates to the overallocation factor discussed above. The original C version `malloc`'s these maximum-sized arrays and initializes only the portions of these arrays corresponding to the original mesh. The Java procedural version mimics this behavior, but is forced by Java language semantics to initialize the entire array with the appropriate default values. The O-O version constructs only the array of references to the fully refined mesh and the geometric objects corresponding to the original mesh. However, this is a zero-sum game, and the cost of constructing the rest of the geometric objects in the O-O version is accounted for in the adapt time.

The large number of objects can be avoided by encapsulating the lists of vertices, edges, and elements into *VertexList*, *EdgeList*, and *ElementList* classes in which the individual fields from the *Vertex*, *Edge*, and *Element* classes are each grouped into their own array. We did not try this technique, as it limits code reuse and extendability and would not be considered good object-oriented practice. However, it does represent an intermediate version between our procedural and O-O versions and would most certainly reduce GC time.

As stated in Section 3.2, we initially tried an O-O version that used *ArrayList* objects from the Java Collections framework [30] to manage our lists of geometric objects. We then used iterators to traverse the lists. Although not shown here, this version performed quite poorly on the Pentium. Quantifying the inefficiencies requires further study, but two factors probably contribute: the pervasive use of interfaces (which complicates dynamic method dispatch) and the obligatory conversions to and from the `java.lang.Object` superclass. We note that previous researchers [17, 23] also describe inefficient performance of several of the Java libraries.

4.4.1 Multithreaded Performance

Figure 9 shows the multiprocessor speedup of the O-O Java version of 2D_TAG on the 12-processor PowerPC machine with the Jalapeño JVM. "Locks" refers to the use of `synchronized` methods in the *Edge* and *Vertex* classes to prevent conflicts among edges and vertices that are shared by multiple threads. As might be expected, this strategy performs poorly and little, if any, speedup is realized. In fact, for 8 and 12 threads, the running times are worse than the single thread time. The next strategy, "Static Partitioning", partitions the global mesh into submeshes with METIS[22] and assigns a thread to work on each submesh. Locks are then required

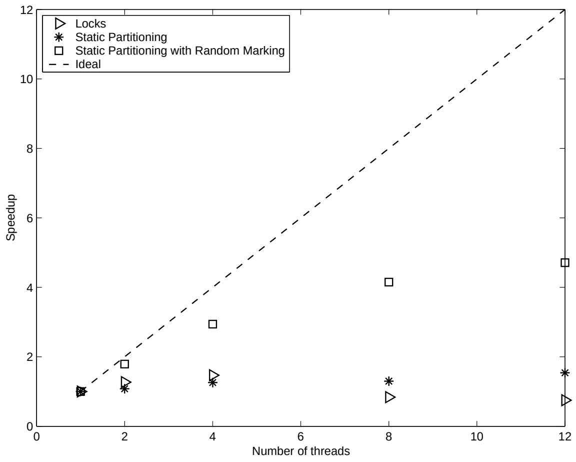

Figure 9: Speedup of 2D_TAG on 12-processor PowerPC with Jalapeno.

for only those edges and vertices on partition boundaries. The static partitioning time is 0.20 *sec* on the PowerPC which is small compared to the startup and adaptation times of 2D_TAG (see Table 8) and is not included in the total running time when computing speedup. The speedup using this strategy is only slightly better than with using locks, reaching a maximum speedup of 1.54 when using 12 threads. The poor performance is not due to excessive synchronization but due to load imbalance. Because we are only partitioning once, the computational load becomes increasingly imbalanced with each level of refinement. The solution is to repartition the refined mesh at each step. To simulate more favorable load balancing, we ran 2D_TAG using "Static Partitioning with Random Marking". Edges are marked for refinement randomly and not according to a error criteria. The submeshes remain approximately the same size during the five levels of refinement, and this fact accounts for the improved speedup, which reaches 4.71 when using 12 threads. Including repartitioning times would increase the running times and reduce these measured speedups somewhat. This is not a realistic scenario, but in the absence of repartitioning, does demonstrate the parallel performance of the multithreaded Java version of 2D_TAG on similarly-sized submeshes.

5. CONCLUSIONS AND FUTURE WORK

We have described the design, implementation, and evaluation of object-oriented "100% Pure" Java codes for two representative CFD applications that differ radically in their characteristics. The flow solver LAURA is structured, largely static, and floating-point intensive. The mesh adaptation algorithm 2D_TAG is unstructured, very dynamic, fine-grained, and limited by the speed of object manipulation. We have demonstrated that LAURA's running time is almost within a factor of 2 and 2D_TAG's is within a factor of 1.5 of their optimized native, procedural counterparts using current off-the-shelf Java compilers and JVM technology. We feel that this level of performance is extremely promising and is susceptible to further improvement as Java technology matures.

Performance should of course be considered in the larger context of the software design cycle. In both codes, the Java versions are smaller in size than the original versions: 5300 lines of Java to 14500 lines of Fortran for LAURA, and 1300 lines of Java to 1900 lines of C for 2D_TAG. In addition, the Java versions are much more modular and hierarchically structured. The Java version of LAURA contains 49 classes organized in a three-deep inheritance hierarchy, while the O-O version of 2D_TAG contains eight classes in a two-deep hierarchy. Such modularization makes it easier to extend the functionality of the code, e.g., to add different boundary conditions, gas chemistry options, or relaxation algorithms in LAURA. Programmer productivity also needs to be considered. The Java versions of both codes were designed and implemented in 18 months by a single programmer with minimal Java experience at the beginning of the effort. The original versions, by contrast, represent many person-years of effort.

The performance of the codes indicates the complex design tradeoffs in JVM implementations and highlights several aspects of JVM behavior that should be investigated further for high-performance Java. First, the user can control garbage collection activity only in very indirect ways, such as by specifying the initial and maximum heap sizes. The wide variety of GC algorithms, their different performance characteristics, and the black-box nature of this component of the JVM, make this level of user interaction inadequate for high-performance applications. Greater user interaction with the garbage collector within the constraints of the Java platform need to be investigated. Second, the performance of multithreading varies widely among platforms, and needs to be improved to scale to large numbers of threads.

Regarding future work, we plan to implement a parallel, three-dimensional version of the 2D_TAG code as well as continue our experimentation.

6. ACKNOWLEDGMENTS

We thank Peter Gnoffo of NASA Langley Research Center for providing us with a copy of the LAURA flow solver and Steve Fink, Peter Sweeney, and Michael Hind of IBM T.J. Watson Research Center for their permission and assistance in running our codes on the Jalapeño JVM.

7. REFERENCES

[1] G. Almasi, F. G. Gustavson, and J. E. Moreira. Design and evaluation of a linear algebra package for Java. In *Proceedings of the ACM 2000 Java Grande Conference*, pages 150–159, June 2000.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '00)*, pages 47–65, Oct. 2000.

[4] R. Biswas and R. C. Strawn. A new procedure for dynamic adaption of three-dimensional unstructured grids. *Applied Numerical Mathematics*, 13:437–452, 1994.

[5] B. Blount and S. Chatterjee. An evaluation of Java for numerical computing. *Scientific Programming*, 7:97–110, 1999.

[6] Z. Budimlic and K. Kennedy. Optimizing Java: Theory and practice. *Concurrency: Practice and Experience*, 9(6):445–463, June 1997.

[7] M. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley.

The Jalapeno dynamic optimizing compiler for Java. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 129–141, June 1999.

[8] H. Casanova, J. Dongarra, and D. M. Doolin. Java access to numerical libraries. *Concurrency: Practice and Experience*, 9(11):1279–1291, Nov. 1997.

[9] F. M. Cheatwood and P. A. Gnoffo. User's manual for the Langley Aerothermodynamic Upwind Relaxation Algorithm (LAURA). NASA TM-4674, Apr. 1996.

[10] G. Fox, X. Li, Z. Qiang, and W. Zhigang. A prototype of Fortran-to-Java converter. *Concurrency: Practice and Experience*, 9(11):1047–1061, Nov. 1997.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[12] P. A. Gnoffo. An upwind-biased, point-implicit relaxation algorithm for viscous, compressible perfect-gas flows. NASA TP-2953, Feb. 1990.

[13] P. A. Gnoffo. Asynchronous, macrotasked relaxation strategies for the solution of viscous, hypersonic flows. AIAA paper 91-1579, June 1991.

[14] P. A. Gnoffo, K. J. Weilmuenster, and S. J. Alter. Multiblock analysis for Shuttle Orbiter re-entry heating from Mach 24 to Mach 12. *Journal of Spacecraft and Rockets*, 31(3):367–377, 1994.

[15] P. A. Gnoffo, K. J. Weilmuenster, H. H. Hamilton, D. R. Olynick, and E. Venkatapathy. Computational aerothermodynamic design issues for hypersonic vehicles. AIAA paper 97-2473, June 1997.

[16] A. Harten. High resolution schemes for hyperbolic conservation laws. *Journal of Computational Physics*, 49(3):357–393, 1983.

[17] A. Heydon and M. Najork. Peformance limitations of the Java core libraries. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 35–41, June 1999.

[18] W. Hoschek. Uniform, versatile and efficient dense and sparse multi-dimensional arrays. `http://cern.web.cern.ch/cern/divisions/ep/hl/papers/ACMJava2000.pdf`, 2000.

[19] S. Itou, S. Matsuoka, and H. Hasegawa. AJaPACK: Experiments in performance portable parallel Java numerical libraries. In *Proceedings of the ACM 2000 Java Grande Conference*, pages 140–149, June 2000.

[20] M. Jacob, M. Philippsen, and M. Karrenbach. Large-scale parallel geophysical algorithms in Java: A feasibility study. *Concurrency: Practice and Experience*, 10(11-13):1143–1154, Sept. 1998.

[21] Java Grande Forum. Making Java work for high-end computing. Panel at SC'98, Orlando, FL, `http://www.javagrande.org`, Nov. 1998.

[22] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. TR 96-036, Department of Computer Science, University of Minnesota, 1996.

[23] R. Klemm. Practical guidelines for boosting Java server performance. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 25–34, June 1999.

[24] R. A. Mitcheltree and P. A. Gnoffo. Wake flow about the Mars Pathfinder entry vehicle. *Journal of Spacecraft and Rockets*, 32(5):771–776, 1995.

[25] J. E. Moreira, S. P. Midkiff, and M. Gupta. From flop to megaflops: Java for technical computing. *ACM Trans. Prog. Lang. Syst.*, 22(2):265–295, Mar. 2000.

[26] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. Java programming for high performance numerical computing. *IBM Systems Journal*, 39(1):21–56, 2000.

[27] L. Oliker and R. Biswas. Parallelization of a dynamic unstructured algorithm using three leading programming paradigms. *IEEE Transactions on Parallel and Distributed Systems*, 11(9):931–940, Sept. 2000.

[28] C. J. Riley and F. M. Cheatwood. Distributed-memory computing with the Langley Aerothermodynamic Upwind Relaxation Algorithm (LAURA). *Advances in Engineering Software*, 29(3–6):317–324, 1998.

[29] P. L. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics*, 43(2):357–372, 1981.

[30] Sun Microsystems Inc. The collections framework. `http://www.javasoft.com/products/jdk/1.2/docs/guide/collections/index.html`, 1999.

[31] H. Yamauchi, A. Maeda, and H. Kobayashi. Developing a practical parallel multi-pass renderer in Java and C++. In *Proceedings of the ACM 2000 Java Grande Conference*, pages 126–133, June 2000.

[32] H. C. Yee. On symmetric and upwind TVD schemes. NASA TM-86842, Sept. 1985.