Heterogeneous Programming with Java: Gourmet Blend or just a Hill of Beans?

Charles C. Weems Jr.
Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610
weems@cs.umass.edu

Abstract

The heterogeneous parallel processing community has long been struggling to bring its approach to computation into the mainstream. One major impediment is that no popular programming language supports a sufficiently wide range of models of parallelism. The recent emergence of Java as a popular programming language may offer an opportunity to change this situation. This article begins with a review of the special linguistic and computational needs of heterogeneous parallel processing by considering the user communities that would benefit most from the approach. It then reviews the pros and cons of Java as a language for expressing and realizing heterogeneity, and concludes with some possible changes that would make Java more suitable for such use.

1. Heterogeneous Programming: Who Needs It?

Before we look at the relationship between Java and heterogeneous programming, we should first review what is involved in programming heterogeneous systems: where are they used and how? Once we identify the requirements for supporting software development for heterogeneous systems, we have a better basis for judging the applicability of a programming language. What follows is not meant to be an exhaustive survey of the field, but merely a discussion of some well-known examples to motivate the identification of a set of requirements.

There are three basic reasons for writing programs that involve heterogeneous parallelism: because we need to use heterogeneous hardware, because our problem is inherently heterogeneous in nature, or because we are faced with some combination of the two. In practice there are many gray areas between these distinctions. For example, to some applications, a distributed shared memory parallel processor may be completely homogeneous, whereas others may be sensitive to differences in memory access time and thus see such hardware as heterogeneous. Likewise, while one approach to solving a problem may be inherently heterogeneous, there may be other approaches that are more homogeneous in nature. In what follows it is implicit that programmers are always faced with a spectrum of choices and that the use of heterogeneity in any given instance is a matter of degree rather than absolute.

2. Heterogeneous Hardware Users

In some situations, the system architect is forced to turn to heterogeneous hardware. The necessity for heterogeneity can be due to space and power requirements as in embedded processing, or due to cost considerations as in clustered workstation farms, or simply a matter of physical limitations of technology as with large-scale shared memory multiprocessors. Heterogeneity can also result from systems that change their configuration dynamically, as in the case of adaptive computing hardware or network computing in which the availability of nodes is subject to change. In the sections that follow, we consider some of the special programming issues that are associated with each of these situations.

2.1 Embedded Systems

Most embedded systems are strongly constrained by limitations such as size, weight, power and cost. Many embedded systems are not high-performance in nature, and the goal is simply to minimize cost while achieving the necessary level of performance. However, when requirements for high performance are combined with embedded system limitations, there is often a considerable benefit to employing heterogeneous parallelism. For example, combining a digital signal processor (DSP) with a microprocessor and some custom logic can be more cost

effective or achieve a higher level of performance than using multiple identical microprocessors.

Achieving high performance with DSP and custom logic, however, involves an especially high degree of optimization of certain algorithms for the hardware. There may be just a single way of optimally coding an algorithm for a DSP that was specifically envisioned by its designer. For example, some DSP architectures include address arithmetic instructions that are unique to a Fast Fourier Transform (FFT), and their use can speed up the inner loop of that algorithm by nearly an order of magnitude. Typically, these algorithms are hand-coded in assembly language and provided as external libraries.

While the library approach works in limited situations, it presents problems of portability and flexibility. One of the goals of heterogeneous programming is to reduce the dependence on hard-coded machine-specific libraries so that code can be ported to different heterogeneous platforms with minimal effort. A program that is written with such library calls can't be ported to another platform (or even run on a uniprocessor) until the library is rewritten for the new platform.

The alternative is that we write the library's algorithms in a high level programming language so they can be compiled for whatever system we choose. Of course, we then generate suboptimal object code for the DSP. While we could perhaps build a compiler to recognize and optimize certain key DSP algorithms carefully written in some canonical form, it would be difficult to handle the broader spectrum of DSP algorithms or even minor variations on the key subset.

A simple but effective solution is to provide the programmer with the ability to uniquely name an algorithm that is implemented in multiple ways (i.e., in high-level code and in libraries) and to indicate either a specific target or the conditions that determine the appropriate target for each implementation. For example, a program might include the code for a generic FFT, and the compiler might detect that there is a corresponding FFT library function for one of the target processors. Depending on how the code is partitioned among the processors, the compiler either generates new FFT code or a library call. Database researchers refer to this as ad-hoc polymorphism, and we have previously called it pseudomorphism [Weems1994] because it is analogous to the mineralogical form of the same name in which a crystal is chemically replaced by another compound in such a manner that the external appearance remains unchanged.

Implicit in the foregoing discussion is the notion that the compiler or some other tool is able to partition code among processors of multiple types. Partitioning implies that there is some means of estimating the performance and cost of mapping code segments to processors. While it is sometimes possible for partitioning tools to analyze code and identify first-order factors affecting its performance, it is also the case that the programmer may have specific information that can help to guide partitioning, and should be given a means to express it.

Partitioning tools also need hardware-specific cost estimators both for the individual target processors and for the communication mechanisms that connect them. This can either be in the form of dedicated software for each target or more general software that bases its estimates on hardware descriptions expressed in some language. This isn't necessarily the same language that the programmer uses, but it is difficult to decide whether it is best to create a whole new language or to extend an existing language with constructs that most programmers will never use.

2.2 Adaptive Computing

Processors that can change their configuration, such as field programmable gate arrays (FPGA) present challenges that are similar to heterogeneous computing systems. They are usually employed in embedded applications where separate processing phases require different custom computing hardware and thus it is possible to use a single component that reconfigures itself between phases. Adaptive hardware is often used as a coprocessor in a system that includes a DSP or traditional microprocessor.

Like DSP systems, adaptive systems often rely on libraries of manually optimized functions. An alternative approach for programming adaptive devices is to generate configurations automatically. Currently this is done only from hardware description languages (HDL) or from customized high-level languages that enable users to express computations in ways that are more suited to hardware layout (e.g., dataflow with datapath width information).

In terms of heterogeneous programming, the implications of the library approach are similar to those for DSP-based embedded systems. However, for automatic generation of configurations, the implications are that a language should provide some features similar to those of hardware description languages, including pipelining, clocking and synchronous communication, datapaths and functional units of varying widths.

The implications of adaptive computing for partitioning and mapping are that the cost model is more complex and performance estimates depend more on detailed analyses of the actual circuitry. Because there are many ways to lay out a particular circuit that affect different aspects of its performance, there is a larger mapping space to explore. The mapping space could be

considerably constrained by additional information from the programmer.

2.3 Clustered Workstations

Heterogeneous computing is often most closely associated with networked workstations in which multiple models are employed so that nodes differ significantly from each other in terms of performance and capacity. In many cases, the workstations are used in a manner similar to a homogeneous parallel processor and it is simply a matter of partitioning operations in parallel across the available resources. The reason for adopting this approach is typically to save cost by using existing hardware resources and free software to build an ad-hoc parallel processor, although some clusters are purpose-built.

Because clusters typically employ a standard computer network, communication between nodes has high latency and limited bandwidth. Thus it is common to partition jobs in a manner that minimizes communication, such as having a master processor distribute work to slaves that compute intensively for some period before returning a result. Partitionings of this nature are naturally expressed via message passing, and a significant amount of legacy code now exists that uses either the PVM or MPI library. Thus, in the near term a language must support interfaces to these libraries.

In the long term, a goal of heterogeneous computing research should be to support more automation of code partitioning, mapping, and distribution in these environments. However, their sheer diversity combined with a focus on low cost and modest software effort may make it difficult to provide a more sophisticated solution that is acceptable to this particular user community.

2.4 Nonuniform Memory Access

As MIMD parallel processors scale up in size, they encounter various physical limits that force their designers to sacrifice uniformity of memory access latency. One approach that has been adopted is to cluster processors in groups of two to eight within which they have uniform access latency, and access to shared memory outside of the cluster is slower (Figure 1). In some cases, the clusters are also grouped into a hierarchy. Another approach is to connect the clusters with a message-passing network with the result that programs can either employ a heterogeneous mixture of shared and distributed memory, or they can use software emulation of shared memory outside of clusters with a resultant increase in latency.

All of these architectures benefit from appropriate partitioning and mapping to enhance locality of reference. Traditional memory placement optimizations can be modified to some extent to deal with the nonuniform access latencies, and in doing so start to resemble partitioning strategies for heterogeneous systems. High Performance Fortran (HPF) is a recent attempt to extend a language with constructs that enable the programmer to provide additional information to aid the partitioning of data.

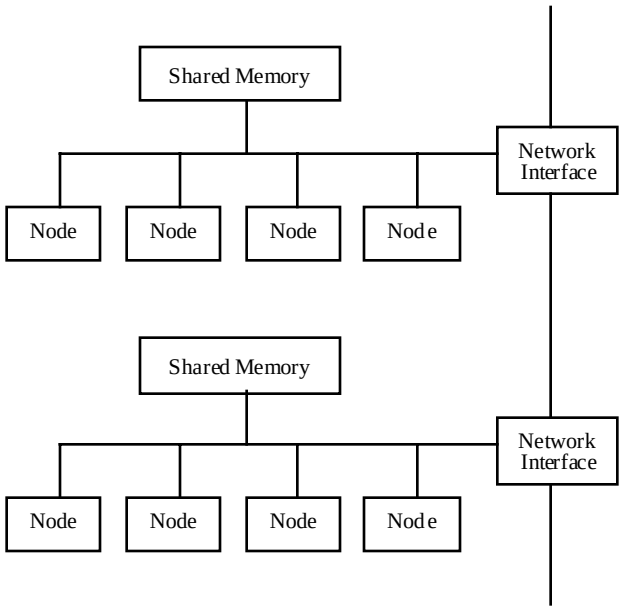

Figure 1. Clustered Parallel Architecture

While some sort of programmer-provided partitioning information is probably needed, the HPF extensions and especially their interactions with the Fortran-90 parallel extensions have proved to be particularly troublesome to compiler writers. Thus, an important consideration in evaluating a language for heterogeneous programming is to ensure that its features do not conflict with each other.

3. Solvers of Heterogeneous Problems

As we move beyond embarrassingly parallel applications to address problems with greater complexity, we find that they are often most naturally expressed with a combination of parallel processing modes. In the previous sections, we focused on the language features that are needed to enable users to inform partitioning tools so that they can distribute code onto heterogeneous hardware. In the sections that follow, we consider via some examples the language features needed to express the natural heterogeneity of parallelism in some applications.

3.1 Scientific Computing

Scientific codes are relying more on irregular computations, as in the case of simulating turbulent fluid flow, in which areas outside of turbulence zones are regular and sparse but within turbulence zones meshes are dense and irregular in their topology. Applications such as these require language features that support the definition of appropriate data structures, such as variable density triangular meshes, and new arithmetic operators that process them. One approach to implementing these is to use an index array that points into a data array and explicitly manage the adjacencies in the mesh. A set of functions can be written to carry out arithmetic operations on the mesh. However, all of this structure obscures the true relationships between the data elements and between operators, making it difficult to optimize them.

Expressing an irregular data structure at a higher level should both simplify programming and optimization. For example, making the property of adjacency explicit might enable programmers to define an adjacency matrix and automatically inherit certain operations that depend on adjacency. While this could be syntactically expressed with a class hierarchy, the compiler sees only the underlying implementation and cannot take advantage of higher level aspects of adjacency as a property in optimizing computations.

Linguistically, the ability to overload arithmetic operators provides a convenient syntactic sugarcoating that hides the functional implementation of operations on new data types. But overloading by itself does not enable the compiler to optimize expressions of these new operators in the same way that it optimizes expressions made up of built-in operators. The ability to add semantic property information to definitions to facilitate optimization is a necessity. Such a capability need not resort to the generality and complexity of mechanisms such as denotational semantics [Gordon1979]. Rather, simply having a list of properties that the compiler recognizes and which can be attached to definitions would be adequate.

Simulations of whole systems or processes are a natural source of heterogeneous parallelism. For example, simulating an entire jet engine involves various stages of compression, fuel-air mixture, combustion and exhaust, as well as mechanical stresses and thermodynamics. Simulating each of these aspects of the engine involves different computational techniques with different degrees of parallelism, connected in a dataflow structure that mimics the physical relationships between the engine components.

3.2 Computer Vision

The complexity of interpreting visual information necessitates many different processing techniques. Image processing and feature extraction provide opportunities for fine-grained data parallel processing. Image sequences can be processed in a pipelined manner and multiple features can be extracted simultaneously with MISD parallelism. Extracted features can be combined with SPMD parallelism into larger structures, and shared memory MIMD parallelism can be used to search the extracted features for matches to multiple objects at once. The overall processing may be coordinated with a data flow model.

The implications of our two example application domains for a programming language are that it must be able to support a wider range of modes of parallelism than merely data parallel and multiprocessing in order to facilitate programming of heterogeneous applications. In addition, it must support novel combinations of parallelism. Ideally, it would also enable the programmer to define new models of parallelism to suit a specific problem.

4. Summary of Requirements

From the foregoing discussion, we list the following requirements for a programming language suited to heterogeneous programming:

- The ability to uniquely identify algorithms.
- Express partitioning and mapping information.
- Interface to PVM, MPI, etc.
- Support message passing over networks.
- Avoidance of conflicting language constructs.
- Overloading of arithmetic operators.
- Ability to add semantic information.
- Ability to define irregular structures.
- Express structural relationships such as adjacency.
- Support a wide range of parallel models, including
    - SIMD,
    - SPMD,
    - MISD,
    - MIMD,
    - shared and distributed memory,
    - dataflow,
    - pipelining.
- Allow parallel models to be combined flexibly.
- Permit extension to new models of parallelism.

In our previous survey of programming languages for heterogeneous parallelism [Weems1994], we identified a

set of requirements that make up the abilities to flexibly combine models of parallelism into new models:

- Support for a range of data and control grain sizes.
- Ability to define a communication abstraction.
- Ability to define a synchronization abstraction.
- Ability to specify patterns of data distribution.
- Ability to specify patterns of process distribution.
- Able to define new first-class types.

5. Why Java?

5.1 It's Hot.

At one time, introducing a new programming language was just a matter of creating a compiler and making it available. In the world of computing today, where legacy code and compatibility tend to dominate the economics of software development, it takes a confluence of many factors to enable a new programming language to enter the mainstream. Java appears to be the right language to emerge at the right place and the right time.

Because Java is rising fast but is also still evolving, it presents an opportunity for many special interests to try to influence the design of what may become the dominant programming language of the next decade. The parallel and heterogeneous computing communities have been struggling for years to develop languages that would gain some measure of acceptance. If Java can be made suitable for their purposes with just a few minor additions, then perhaps those approaches will finally enter the mainstream and researchers can move on to new levels of research. Java was originally designed for embedded systems, so it would superficially appear that it should address the concerns of that segment of the heterogeneous processing community.

5.2 It's Simple

In comparison to other modern object-oriented languages, the essential syntax of Java is reasonably simple, and it's core is familiar to any C programmer [Gosling1996]. It avoids many of the pitfalls of larger languages like C++ and Ada95 but does not sacrifice convenience for the sake of minimalistic purity. It may certainly be argued that there are syntactically "better" languages, but there is general agreement that Java is an improvement over many of its predecessors.

Simplicity also implies that a language will be easier to learn, Java thus has the potential to win over converts from existing languages used in heterogeneous, parallel, embedded, and adaptive programming such as Fortran, C, C++, Ada, and VHDL. Of course, to do so, it must also provide a similar level of convenience of expression of essential constructs in each of those domains.

Being syntactically simple, Java offers a better base upon which to add constructs in support of heterogeneity, if necessary, because the resulting interactions between new and old constructs are easier to enumerate. Java's simplicity includes the avoidance of many features of earlier languages that were machine-dependent or that gave the user too much low-level control (e.g., common, equivalence, pointer address arithmetic, explicit memory management). It thus offers a stronger foundation for extensions.

5.3 It's Portable

Because Java targets a virtual machine (the JVM [Lindholm1997]), it can be executed on any machine with an implementation of the JVM. This approach to a run-time environment overcomes many of the difficulties that are faced in trying to distribute code across heterogeneous systems. Issues of word size, endianness, register file size, operating system, and display environment all disappear. In effect, Java homogenizes the heterogeneous world for us through its virtual machine.

The result of this homogenization is that we can focus on the bigger picture of high-level performance characterization in support of partitioning and mapping. All code is generated for a single instruction set and descriptions of target machines can be in simpler terms such as memory capacity and performance on some Java benchmark kernels.

5.4 It's Object-Oriented

Besides being a popular buzzword, object-orientation gives a language great syntactic power for extension. It is possible to develop classes that provide convenient abstractions for many forms of parallelism. For example, one could build an irregular array class that makes it simple to express the kinds of computations that are performed in turbulent fluid-flow simulations. That abstract interface can hide either a sequential or a parallel implementation.

Object orientation also offers greater potential for reuse and extension of user constructs by others. Given classes that implement different models of parallelism, we could combine them in a variety of ways to enable heterogeneity to be expressed.

5.5 It's Garbage Collected

By avoiding explicit memory management by the programmer, Java greatly reduces the opportunity for

errors resulting from memory leaks or bad address arithmetic. However, for parallel processing, garbage collection offers an opportunity for redistribution of processing load. As a garbage collector sweeps through memory compacting live and dead values, instead of merely gathering the live values into one place it could redistribute them to other (possibly heterogeneous) computational nodes.

5.6 It's Threaded

Java's threads provide a simple but powerful mechanism for expressing shared-memory parallelism. Many models of concurrent programming can be expressed with the Java's threads [Lea1997]. It's runtime system, the JVM, already supports multiprocessing so that a consistent abstract parallel environment can be used on all targets.

5.7 It's Network Aware

Java applets provide a means for distributing code across a local area network. In addition, its object-oriented approach can be used as a form of message passing through the remote method invocation facility. More directly, it has the ability to explicitly handle network connections as streams. When this is combined with Java's reflection capabilities, we find that there are many options for implementing parallelism in Java.

6. Java Weaknesses

6.1 It's Hot

The fact that Java is new and being influenced by many interest groups means that the language itself is continuing to evolve at a fast pace, making it difficult to identify specific extensions without risking conflicts that could arise from other proposed extensions. Java may still evolve into a language with features that make it unsuitable for heterogeneous parallel applications. Given the ongoing battle between Microsoft and Sun over Java standards, it is even still possible that Java will fail to reach a critical level of acceptance or be replaced by a different language.

Because of Java's perceived popularity, the heterogeneous programming community has very little influence over Java's evolution, and so must ride the coattails of others in any effort to extend Java for their benefit. A domain-specific language would not be subject to the same restrictions.

6.2 It's Simple

One of the key simplifications that the Java designers chose was to avoid operator overloading. Unfortunately, the lack of operator overloading precludes programmers from writing new arithmetic classes that are as convenient to use as the built-in arithmetic types. Without operator overloading, it will be difficult to convince scientific and engineering users of languages like Fortran 90, HPF, C++, and Ada to switch to Java. In effect, they must return to the levels of notational convenience that were available in Fortran 77, C and Pascal.

While Java provides access to the IEEE floating point standard's special values of plus and minus infinity, plus and minus zero and not-a-number, it does not provide the ability to control the various states of the IEEE standard, such as the rounding mode or the use of subnormal values. Java does not generate any exceptions for floating point operations, and for integers the only exception is division (or remainder) by zero. In particular, overflow and underflow are not detected. These limitations do not help to convince programmers of numerically-oriented applications to switch to Java.

6.3 It's Portable

Java's portability comes at the price of another layer of abstraction (the JVM), which reduces performance. If the abstraction layer is implemented by a bytecode interpreter, then the cost is quite high over native code execution, both in terms of time and space. If JIT compilation of the bytecodes is used, then the time penalty can be reduced as long as the compilation time can be amortized over enough execution time for each run. Off-line translation of bytecodes into native code avoids the JIT compilation cost on each run, but bytecodes are then effectively a low-level intermediate representation of the program that prevents the translator from applying higher-level optimizations before generating code. The result is object code that is less optimized than a native compiler could produce from source code.

Java's approach to portability thus results in a significant reduction in performance. This should be anticipated simply from the way in which Java attempts to homogenize the world of processor architectures. There is little point in using heterogeneous hardware if it will be programmed in a manner that ignores the special hardware features that were selected to improve performance. Java's bytecodes were designed in part to facilitate embedded processing such as appliance controllers, but not high-performance embedded processing such as DSP.

6.4 It's Object-Oriented

Object orientation provides only the illusion of language extensibility. However, there is still a rigid distinction between the first-class constructs and types of the language and user-defined objects. First-class objects carry additional semantic properties that are used for key optimizations. The programmer has no way of providing similar semantic information when defining new objects. The programmer cannot even indicate that existing semantic properties associated with first-class types apply equally to a new type or operator. For example, in defining a new numeric type, the programmer has no way to indicate that the addition operator is associative.

While object oriented programming can be used to define a triangular mesh data type that syntactically simplifies the coding of applications, the compiler has to optimize at a much lower level of abstraction and will miss opportunities for high-level optimizations. This is problematic for partitioning and mapping in heterogeneous systems, where knowing the organization and access patterns of larger structures is especially valuable.

6.5 It's Garbage Collected

The time-penalty of garbage collection is difficult to predict. Depending on the available system memory and on the data being processed, the frequency with which the garbage collector is invoked and the time that it takes can vary considerably. In many systems, long pauses for batch collection are unacceptable, but incremental garbage collection can be used to distribute the time so that the user does not notice delays but merely sees some variations in performance.

In hard real-time embedded systems, however, the performance of the processor must be highly predictable to avoid missing deadlines. The alternative is to assume a worst-case execution time that is so great that much of a processor's performance goes unused because tasks are scheduled under the assumption that they will all suffer a maximum penalty for garbage collection.

6.6 It's Threaded

While threading provides at least one native model of parallelism in the language, it is not sufficient by itself for efficiently defining all other models of parallelism. Combining threads with object-oriented techniques and other features of the language could make it possible to syntactically express other models at least in a round about way. For example, SPMD parallelism could be crudely implemented with an array of threads, onto which data arrays are partitioned, and locks could be manually programmed to ensure synchronization at the end of each basic block. However, that is far less convenient than a FORALL or PARDO construct.

The appearance of SIMD parallelism could be obtained with suitable whole-structure operators on structured classes. But the only actual means of implementation would be external library calls. Without operator overloading and semantic extensibility, however, there would be no opportunity for optimizing across calls. For example, an expression (written as method calls) that operates on a parallel array type could not have common subexpressions eliminated, nor could the registers of the parallel hardware be scheduled efficiently.

In essence, Java provides one basic approach to parallelism, and does not facilitate the use of other models. It thus neglects the needs of people who need to solve heterogeneous problems. It has also been noted that the primitive nature of Java's threading facility is analogous to the combination of pointer arithmetic and explicit memory management in C in that it provides sufficient rope to hang the programmer [Lewis1997]. Various user communities would be better served by a parallelism facility that is less prone to deadlock, livelock, and orphaned threads.

6.7 It's Network Aware

Java's network awareness is built on top of TCP/IP and HTML. Connections are established between Java programs using internet addresses and port numbers, and applets are referenced by internet addresses plus file names. While this greatly simplifies the network interface and makes it widely portable, the latency of the layered network protocol is significant. For a cluster of workstations, where the communication network is also built around this protocol, there is little choice but to accept the high latency. However, in purpose-built clusters or distributed-memory parallel processors, where communication is mainly between trusted peers, the latency is unnecessary.

While this is partly an operating system issue, Java does not inherently provide a mechanism to distinguish between secure communication and trusted low-latency communication. Adding a library built on special OS calls would allow programmers to work around this limitation, but then their code would not be portable.

There are no provisions in Java's model of network communication to support parallel operations such as broadcast and reduction. Again, these could be provided with manual workarounds, such as a native interface to MPI, but the bottom line is that Java was not designed for parallel processing network communication.

7. Making Gourmet Java

The foregoing discussion highlights various features of Java that make it both attractive and unsuitable for heterogeneous programming. In this section we consider some possible changes that would make Java more suitable for heterogeneous programming. It should be noted that these are not all syntax changes. Some of the changes involve the compiler and the virtual machine.

7.1 Syntactic Additions

Operator overloading is a key syntactic addition that is needed to enable programmers to write new numeric classes that match the expressiveness of other languages. Syntactically this is a minor change to the language if it is restricted to overloading the existing operators, although it significantly affects parsing, conversions, and promotions. If generalized to enable arbitrary monadic and dyadic operator forms for method invocations, then it would be a more significant syntactic change.

To support pseudomorphism, the syntax for the implements clause in a class declaration should be extended to support multiple implementations of the same interface by classes with the same name in a single package. The redundant classes must then be distinguished by a predicate. For example, we might write

implements *interface-list* when *Boolean-expression*

to indicate the conditions that determine when a particular class from the set of identical classes is selected as the appropriate implementation of an interface.

The conditions could be resolved at compile time, load time, or run time, according to the information that they depend on, which implies that the compiler, loader, and JVM must all be extended to perform these tests in a heterogeneous environment. For example, a condition might call getProperties to determine the type of target onto which it is being loaded so that only the implementations appropriate to the target would be loaded onto it. If multiple implementations are loaded at run time, then when the first member of the interface is invoked, the run-time system must test their conditions to identify the one that will be used. Once an implementation has been selected, it is used for as long as the class instance exists. If the conditions for multiple implementations are satisfied then the run-time system chooses the implementation. If none of the conditions are satisfied, then an exception is thrown.

If conditions change such that it would be desirable to switch to a different implementation (e.g., due to changing system load), then the class instance must be destroyed and a new instance created using the alternate implementation. It would be up to the programmer to decide how to convert the state of one implementation into another in such a transition.

7.2 Semantic Extensibility

A limited but sufficient mechanism for semantic extensibility could be achieved by enriching the set of attributes explicitly recognized by the Java compiler and virtual machine, and making them accessible at the source level. The existing attribute facility of the virtual machine is sufficient to tag any class, field, or method with additional information. For example, the ConstantValue attribute indicates to the JVM that a field is a constant.

If the list of attributes is extended to explicitly include all of the currently implicit semantic properties used to trigger or enable optimizations, and these are made available to the programmer, then it becomes possible to extend the language with new first class types. For example, we might write the following form (which also assumes an extension for overloading arithmetic operators):

```
attributes(associative, commutative)
static complex dyadic +
                (complex left, complex right)
```

This code would define a new + operator for a complex type that would carry the built-in attributes necessary to enable high-level optimizations of expressions containing it.

If the user specifies an attribute that does not exist in the system, a warning would be issued, but it would not result in a fatal error. The JVM is specified to ignore unrecognized attributes. This would allow implementations to carry attributes that are specific to certain JVMs but not others (e.g., a parallel JVM). In addition, because the attributes are carried through to the bytecode representation, it is possible for a bytecode to native code translator to employ some of its own high-level optimizations. For example, the translator could use the attributes to enable more aggressive register scheduling.

One attribute that would facilitate the creation of irregular types would be a means of indicating the adjacency relationships between elements in a data structure. Attributes could also be used to carry information to guide partitioning and mapping of structures.

7.3 Compiling

Compiling Java in a heterogeneous environment is likely to involve the initial bytecode generation, together with native code generation (either off-line or JIT) once the bytecodes are downloaded to a specific target. As mentioned in the previous section, a wider range of attributes must be carried in the class file containing the bytecodes to enable delayed optimization. It may, in fact, be necessary to carry a higher-level intermediate representation (IR) of the code in the class file to enable all of the desired optimizations. The IR might end up being comparable in sophistication to a program dependence graph augmented with source node-type information and links to the generated bytecode. In essence, this would enable the native code translator to start with a higher level view of the code and generate all new code for the particular target. Having such information available would especially facilitate target-specific partitioning of parallel operations.

7.4 Garbage Collection

Currently, Java supports the gc method to force garbage collection to occur. However, there is no way to ensure that collection does not occur during a time-critical section of code. The gc method should be extended to accept various parameters, such as gc(FALSE) to turn off collection until it is either explicitly enabled with gc(TRUE), or the current method exits.

In addition, the user could be given more control over the collection process, such as indicating whether a more costly incremental collection should be run to minimize pauses or that a faster periodic sweep that results in noticeable pauses is acceptable.

7.5 Other Models of Parallelism

There are other extensions that would make it easier to explicitly write certain kinds of parallel code, such as FORALL or WHERE, but it can also be argued that parallelizing compiler technology can often identify these cases when normal loops are written to directly implement sequential versions of them.

Directly supporting multiple models of parallelism in Java depends not so much on changes to the language as to the JVM, which currently recognizes only threads as being concurrent. Given operator overloading, pseudomorphism, and semantic extensibility, we can express nearly all forms of parallelism. For example, the features of a data parallel language like ZPL [Snyder1994] can be realized with object-oriented techniques, although not with the same syntactic simplicity. Consider that ZPL regions can be implemented as arrays with the appropriate attributes and region specifiers can be written as methods that manage a global context variable. The result will not be as pretty, but if it carries the appropriate attributes through to a parallel runtime environment, the resulting code should have similar efficiency.

Of course, the ultimate in extensibility would be the capability to define new control structures. However, such an extension would involve the ability to pass expressions and code blocks to methods, where they could be executed with some level of intervention. When combined with support for pseudomorphism, however, the result would be the ability to directly express parallel operations whose implementation is determined by the available hardware; which is precisely what users of heterogeneous processing are seeking.

8. Conclusion

Java offers a combination of opportunities and features that make it an attractive language for heterogeneous parallel processing. However, a deeper study reveals some serious shortcomings that will make it difficult to attract users from the major communities that need heterogeneity. In particular, it lacks key support for scientific and high-performance embedded processing. In addition, the way in which it homogenizes the world to achieve portability directly conflicts with the fundamental reason for employing heterogeneity.

However, with some modest extensions to the language, and suitable restructuring of the compiler, loader, and run-time system (including native code generation from bytecodes and a higher level IR), Java could be made much more suitable for heterogeneous parallel processing. The major syntactic changes would be to enable operator overloading, extend the implements clause so that multiple classes with the same name can implement an interface in a manner that allows the system to choose between them, and make an enriched set of standard attributes accessible to the programmer. Support for user-defined control structures would be a significant change in syntax and semantics, but would allow expression of parallel operations with the syntactic directness of parallel languages such as Fortran90, HPF, and ZPL.

9. Acknowledgments

This work was supported in part by a grant from the Defense Advanced Research Projects Agency, monitored by the Naval Research Laboratory under contract number N00014-94-1-0742. The author wishes also to thank Eliot Moss, Kathryn McKinley, Steve Dropsho, Brendan

Cahoon, Glen weaver and John Cavazos for their helpful discussions and comments. Some of the ideas presented here are closely related to concepts behind the heterogeneous compiler, Scale, which we are presently building.

10. References

[Gordon1979] Michael Gordon, "The Denotational Description of Programming Languages", Springer-Verlag, New York, 1979.

[Gosling1996] James Gosling, Bill Joy, Guy Steele, "The Java Language Specification", Addison Wesley Pub. , Reading, MA, 1996.

[Lea1997] Doug Lea, "Concurrent Programming in Java", Addison Wesley Pub. , Reading, MA, 1997.

[Lewis1997] Ted Lewis, "If Java Is the Answer, What Was the Question?", IEEE Computer, Vol. 30, No. 3, March, 1997, pp. 133-136

[Lindholm1997] Tim Lindholm, Frank Yellin, "The Java Virtual Machine Specification", Addison Wesley Pub. , Reading, MA, 1997.

[Snyder1994] Lawrence Snyder, "A ZPL Programming Guide", Dept. of Computer Science and Engineering, Univ. of Washington, 1994.

[Weems1994] Charles Weems, Glen Weaver, Steve Dropsho, "Linguistic Support for Heterogeneous Parallel Processing: A Survey and an Approach", Proc. IEEE Heterogeneous Computing Workshop, April, 1994, Cancun, Mexico, IEEE Press, Los Alamitos, CA, pp. 81-88.

Biography

Charles C. Weems, B.S. 1977 (honors) and M.A. 1979, Oregon State University, Ph.D. 1984, University of Massachusetts at Amherst. Since 1984 he has directed the Specialized Parallel Architectures research group at the University of Massachusetts, where he is an Associate Professor. His research interests include parallel architectures for computer vision, benchmarks for vision, heterogeneous and adaptive parallel architectures, compilation for heterogeneous and adaptive systems, architectural issues for hard real-time systems, and parallel vision algorithms. He led the development of two generations of both the hardware and software for a heterogeneous parallel processor for machine vision, called the Image Understanding Architecture, under DARPA support. He is the author of numerous technical articles, has served on over a dozen program committees, is chairing the 1997 IEEE International Workshop on Computer Architecture for Machine Perception, the 1998 IEEE Symposium on the Frontiers of Massively Parallel Processing, and co-chairing the 1999 IEEE International Parallel Processing Symposium. He edited special issues of Machine Vision and Applications and IEEE Computer, serves on the board of ACSIOM Inc., is a member of the technical advisory committee to the board of Amerinex Applied Imaging, Inc., and is also the co-author of four widely used introductory computer science texts, and co-edited a book entitled Associative Processing and Processors. Dr. Weems is a member of ACM, a Senior Member of IEEE, a member of the Executive Committee of the IEEE TC on Parallel Processing, the IAPR TC on Special Purpose Architectures, and is an area editor for the Journal of Parallel and Distributed Computing and the SPIE/IEEE series on Imaging Science and Engineering.