Heterogeneous Parallel Computing With Java: Jabber Or Justified?

H. G. Dietz

Purdue University, School of Electrical and Computer Engineering
West Lafayette, IN 47907-1285
`http://dynamo.ecn.purdue.edu/~hankd/`

Is Java a good language for programming heterogeneous parallel computing systems? It is a well-designed modern language that, combined with the Java Virtual Machine (JVM), offers a myriad of modern programming features and excellent portability. However, in speedup-oriented heterogeneous computing, our primary concern is obtaining the best possible execution speed from the heterogeneous system. This paper briefly discusses what heterogeneous parallel computing is really about, lists some of the key features of Java, and finally summarizes how well Java matches the task of programming for heterogeneous parallel computing.

1. What Is Heterogeneous Computing?

Heterogeneous computing refers to the concept of using a collection of machines, in which each machine may have properties somewhat different from the others, to achieve speedup on a computation.

1.1. Architecture

Generally, a heterogeneous collection of machines will be arranged as either a cluster or a group of networked computers.

A **cluster** is a parallel system whose component computers are both physically and logically near each other. For example, a group of workstations and supercomputers within a single facility, all connected by SCI, HiPPI, Myrinet, PAPERS, or other high-performance networks, would be a typical cluster configuration.

The alternative is a more loosely connected group of **networked computers**, in which communication between machines is possible, but perhaps very indirect, with limited bandwidth and high latency. The largest example of this type of heterogeneous system is the Internet, although many groups of computers connected by LANs (local area networks) are also best viewed in this way.

Both arrangements of machines are possible and useful, but the focus is different. Machines in a heterogeneous cluster can truly cooperate, whereas the loose coupling of networked computers generally requires that machines be able to work independently for relatively long periods of time. Note that the time a processor can operate independently is related to how much memory space is available — having more memory frequently allows local buffering of data to be substituted for some communication operations.

Another significant difference is that clusters are generally assembled by design, whereas networked computers are often simply whatever machines happened to be available. Heterogeneous clusters tend to have heterogeneity because it is directly useful; for example, integrating a SIMD supercomputer with a shared-memory MIMD supercomputer so that different portions of a parallel program can each be executed using the parallel execution model that yields the best speedup. In contrast, heterogeneous networked computers often are workstations from several not-quite-compatible vendors, with essentially the same execution model and only minor performance variations.

1.2. Speedup

In a heterogeneous system, speedup can be achieved by two separate mechanisms. **Parallelism across machines** achieves speedup, ideally proportional to the number of machines, by simultaneously executing portions of the computation simultaneously on different machines. However, speedup also can be achieved by increasing the appropriateness/efficiency of the mapping of the computation onto the special abilities of each machine. In most cases, this centers on use of **parallelism within each machine**.

Clearly, the nature of the hardware heterogeneity places emphasis on one or the other of these two mechanisms. Performance of a cluster containing a few parallel supercomputers will critically hinge on the effective use of parallelism within each machine; performance of a network mixing comparable DEC, HP, Sun, SGI, etc., workstations will just as critically depend on parallelism across these machines. Of course, failing to use all appropriate machines or failing to use each machine efficiently lowers performance no matter what structure the heterogeneous system has.

1.3. Portability

Because all we care about is being able to execute the appropriate portions of a program on each machine, and there is nothing (excluding development and maintenance time and cost) to prevent us from writing code specifically tailored to each machine, portability simply is not a requirement. Indeed, if the machines are heterogeneous in the sense that some machines have access to specialized I/O devices that others cannot access, portability is meaningless: running code ported from a computer-controlled milling machine on a workstation is unlikely to get any parts milled. It can be just as difficult to achieve good results when porting a SIMD-optimized algorithm implementation to a MIMD machine.

Accepting that portability is not required, it is a highly desirable goal that all programs be expressed in portable notations. Further, if a program is primarily concerned with "pure" computation rather than I/O, the goal of portability can be achieved. There are two basic approaches.

Portability by simulation: typically, this is done by compiling each program to an idealized, simplified, architecture which is in turn simulated by software on the target machine. This approach first became widely accepted in the form of P-Code implementations of Pascal. In fact, the Java Byte Code and Java Virtual Machine have many striking similarities to the UCSD Pascal P-System (which also incorporated graphics support and a protected operating environment).

Portability by transformability: instead of simulating another target, one can use a combination of compiler analysis and transformation technologies to literally re-write and optimize the program for the specific features of the target machine. Transformation is more difficult to implement than simulation, but the benefit is higher performance.

As a general rule, simulation works best when the idealized architecture has data types and other basic properties that are very similar to the actual target hardware characteristics, but "instructions" that are typically much higher-level than individual target machine instructions. The need for basic properties to be similar is obvious; the need for higher-level instructions is due to the fact that overhead to decode and prepare each simulated instruction for execution generally is typically about 20 target machine instructions. Simulating a "matrix multiply" instruction easily hides a 20-instruction overhead, while simulating "32-bit integer add" may result in a 20x slowdown relative to executing the operation transformed directly into native target code.

Many simulators now incorporate incremental compilers that can perform some of these transformations, however, the apparent simplicity of transforming an add instruction into machine code is misleading. The major complication is essentially the use of parallelism.

1.4. Transformability

If the simulated instructions are simple and not parallel, but the target machine is parallel, we are confronted with the age old problem of automatically parallelizing. Perhaps even more complex is the problem of transforming parallel instructions into a different target parallelism. There are at least three key aspects of a parallel execution model that must be matched when generating code: execution mode, communication model, and grain size/data layout.

1.4.1. Execution Mode

SIMD, MIMD, and VLIW/Superscalar execution modes each perform best under different circumstances, but it is very difficult, for example, to transform arbitrary MIMD code into efficient SIMD code.

SIMD (Single Instruction Stream Single Data Stream), which is now *the most common parallel execution mode* (what do you think all those multimedia enhancements are?), is very effective in implementing algorithms that require tight synchronization of similar activities across many processing elements. For example, communication operations do not need buffering, interrupts, etc., because it is trivial for all processing elements to know precisely when data exchanges will occur. However, SIMD serializes operations that are different on each processing element.

MIMD (Multiple Instruction Stream Multiple Data Stream) is the most general parallel execution mode, capable of simultaneously executing arbitrary operations on different processing elements. Because each processing element can have its own independent clock and program counter, processing elements do not have to wait for the slowest to complete each operation before advancing to the next. However, this independence comes at the expense of the ability to efficiently, globally, coordinate actions as we described for SIMD communications.

VLIW (Very Long Instruction Word) and **Superscalar** execution models are logically somewhere between SIMD and MIMD, offering more generality than SIMD while preserving the ability to globally coordinate parallel actions. The problem is that these techniques require structures that do not scale well, so parallelism width is generally limited as compared to SIMD or MIMD models.

1.4.2. Communication Model

There are at least three fundamentally different classes of communication models, and the best choice for each algorithm or target machine varies.

A **shared memory** model communicates using what appear to be simple memory load/store operations, but there are many flavors differing in how much of memory is shared by which processing elements (shared everything vs. shared something), access time as a function of address reference pattern (logical, physical, and cache structures), and even rules for atomicity and coherence (what references are atomic, how are access races resolved). These complications make shared memory models the most difficult to use efficiently and correctly, and also the most difficult to transform into other communication models — even into other shared memory models. Unfortunately, these are also the most efficient models for many machines.

A **message passing** model creates, sends, and receives messages, generally with one sender and one receiver for each message. The key advantage in message passing systems is the ability of a single message to contain a large data payload; this makes it more effective than shared memory models in utilizing reasonable-bandwidth, high-latency, interconnection mechanisms like UDP or TCP protocols over fast Ethernet. There are also various flavors of message passing, differing primarily in the types and lengths of messages allowed and the possible orderings of sends and receives.

An **aggregate function** model, unlike the other two models, allows any number of processing elements to directly participate in each communication operation. The simplest example is a barrier synchronization, in which no processor enters the next phase of a computation until all have signaled completion of the current phase. More complex aggregates include "collective communications" such as permutations, multi-broadcasts, personalized all-to-all, associative reductions, scans (parallel prefix operations), voting operations, etc. Because aggregates are N-ary operations, whereas shared memory and message passing operations tend to be point-to-point, aggregates offer much better performance on hardware that supports them... aggregates also are the key to efficiently implementing SIMD and VLIW execution modes on what would generally be considered MIMD hardware.

Not only is it difficult to transform between these three different classes of models, but it also can be difficult to convert between different models within the same class. For example, it can be surprisingly complex to convert between two shared memory models that differ only in atomicity and coherence properties.

1.4.3. Grain Size And Data Layout

The amount of computation that each processing element will do between communication operations, and the layout of data structures across processors to make data accesses within a granule local, vary widely depending on the target machine, and transformations are difficult. For example, the HPF (High-Performance Fortran) effort was largely driven by the desire to have programmers specify the data layouts — compiler technology to automatically pick the best layout, or even to efficiently transform between specified parallel layouts, is still under development.

1.4.4. Semantics Enable Transformation

In summary, it is very difficult in the general case (perhaps impossible) to transform code written in one form into efficient code in a different form. Put another way, the above are all really **execution model** characteristics. A **programming model** should be designed to use only constructs that have known efficient implementations for a wide range of execution models.

For example, consider an abstract parallel `if` statement:

```
if (parallel_expr) {
    then_action;
} else {
    else_action;
}
```

SIMD semantics would specify that *then_action* would execution before *else_action* in the case that *parallel_expr* was true for some processing elements and false for others. This implies that side-effects, such as communications in each actions, are strictly ordered with respect to each other, and thus are inherently race-free.

In contrast, MIMD semantics would permit *then_action* and *else_action* to execute simultaneously, thus requiring other mechanisms to enforce ordering of communications within the actions.

The point is simply that, to be efficiently transformable, a parallel language (or programming model) must facilitate transformation into efficient code for any target by using **semantics that are consistent with all possible target models**. Thus, our parallel *if* statement should have semantics that *allow* both actions to occur simultaneously, but do not require this.

The result is that when compiling for a SIMD target, there may be redundant synchronizations in the source program (that were placed there to enforce communication ordering for MIMD execution of the actions), but these are easily mechanically removed by existing compiler technology. Thus, careful selection of transformable parallel semantics is the key to making a parallel programming model directly support heterogeneous supercomputing.

2. Java

Java is a well-designed modern language that, combined with the Java Virtual Machine (JVM, also known as the Java byte code interpreter), offers both flexibility and portability. There is a lot to like about Java, and about its byte code (which, for analysis and transformation purposes, is nearly equivalent to the Java source code).

2.1. Data Types

Java specifies the size of basic data types: `byte` is -128 to 127, `short` is -32768 to 32767, `int` is -2147483648 to 2147483647, `long` is -9223372036854775808 to 9223372036854775807, and `char` is 0 to 65535. This is a great benefit for heterogeneous computing in that it removes precision differences from our concerns in using a variety of machines. Although the lack of unsigned types is somewhat disturbing, they are effectively supported by unsigned operators: `>>>` is the unsigned shift right operator.

Another benefit in Java's type handling is the use of IEEE floating point features such as NaN (Not-A-Number), Infinity, and +/-0 instead of exception mechanisms. The need to create a valid state when an exception occurs is a subtle, yet serious, constraint on the compiler's ability to transform code for parallel execution; Java's definition removes this problem.

Java's high-level handling of arrays as single objects, and the deliberate omission of C-style pointers, make data alias analysis significantly easier for typical constructs than it is for C. The handling of object allocation/deallocation is also cleaner, although the garbage collection scheme significantly complicates the runtime environment and may seriously degrade performance using threads.

2.2. Object Oriented

One of the most praised features of Java is its support for object-oriented programming. In many ways, this is a useful abstraction, but object-oriented indirect function calls make static analysis for parallelization more difficult and less effective.

Java further complicates matters by allowing functions to be specified in a way that facilitates using independently-developed binary code modules together within a program. This very late binding of function calls to code makes it very difficult for static compiler analysis to perform global optimizations, such as those needed for some types of parallelism transformations (e.g., conversion from MIMD to SIMD). Just-in-time or other incremental compilation technologies can help in this respect, but these are not really good solutions, largely because the analysis would be repeated at runtime for each new run by each machine. It is also important to note that, for example, a SIMD supercomputer, which seriously needs the global analysis and transformation, may be a totally inappropriate machine on which to execute the analysis and transformation compiler code.

2.3. Threads

Java directly supports parallel execution using a built-in version of threads. Although Java threads essentially implement a shared memory model, unlike most thread libraries, Java precisely specifies how threads distinguish between actions taken on a thread's local working memory and effects on main memory. The result is a model somewhat more flexible with respect to ordering of accesses, thus hopefully more efficiently implementable on a variety of other shared memory mechanisms. As in C, `volatile` attribute is provided to enforce the programmed ordering of accesses.

In comparison to thread libraries, Java provides higher-level synchronization primitives that take advantage of the difference between a library call and a language construct. The standard lock management to ensure that only one thread is allowed to be within a particular segment of code at any given time (mutual exclusion) is remarkably clean in Java:

```
synchronized(t) {
    // exclusively executed code
}
```

This construct not only manages the locking at entry and unlocking at a normal exit of the segment, but also correctly handles unlocking for any other type of exit from that segment.

2.4. Java, The Standard Language

One of the best aspects of Java is the speed with which a precise standard definition of the language has emerged. There is currently an international standardization effort in progress.

The reason that the standard has moved so quickly is because Sun is essentially in full control. As a general rule, international standards are not created by a single for-profit company, but by non-profit groups; when the vote was taken in late 1997 on formation of an ISO standard for Java, the United States cast its vote against the standardization effort largely because Sun is placed in a controlling position. In fact, Sun even retains all rights to their trademark on the name Java. The vote passed, although the US was not alone in its concern about Sun's control of the standard.

Despite this tightly-controlled approach to popularizing Java, it is important to acknowledge that Sun has done a very good job thus far. Not only is Java a relatively clean language design, but Sun has employed some of their best people to ensure that the language is a success. Sun is also pushing a 100% Java certification effort to discourage people from using non-standard features.

I like the 100% effort; it is the only way to achieve the portability that Java seeks. However, it is important to note that a program that calls native routines is not 100% Java. In other words, Java does not provide any way to use parallelism other than threads; any other parallelism would violate the 100% certification rules. This is a fundamental problem with respect to using Java for high performance computing.

2.5. Support For Networking

Java supports network communication via sockets, and provides library routines for higher-level protocols such as HTTP and FTP. It is easy to imagine, or to create, pure Java code library functions for higher-level parallel-processing message passing. Unfortunately, the basic socket interface, although portable, is not particularly efficient, especially within a cluster connected by SAN (System Area Network) hardware.

2.6. Graphics Support

Java provides a very strong set of graphics facilities that are remarkably portable, independent of host machine OS, windowing system, etc. Then again, you already knew this from watching a certain little Java Aplet wave at you from within your WWW browser.... Java is an excellent way to graphically present and browse data, even within a heterogeneous parallel computation, provided that the graphics computations are not too intense.

3. Conclusions

Is Java appropriate for speedup-oriented heterogeneous parallel computing? The answer really depends on what type of heterogeneity your target system employs.

The thought of thousands of random workstations, PCs (Personal Computers), and NCs (Network Computers) scattered across the Internet being used as a parallel computer appalls me... but I've actually participated in such efforts. Virtually all of these machines are relatively minor variations on the same uniprocessor 32-bit architecture, heterogeneous primarily in the sense that some level of product differentiation is important in establishing a vendor's marketing strategy. The same Java code easily runs on all these machines, and it is relatively simple for the Java code to send an occasional message from one machine to another.

However, the Java byte code will be interpreted very slowly, and with variable and difficult to predict performance, on most machines. Of course, a factor of 20x slowdown on each machine can be repaired by simply using 20x more machines... if you have enough parallelism and machines available. Still, more parallelism means more communication unless you can buffer nearly all data in local memories, and Java communication is also slow. In my opinion, the class of applications for which one should be able to achieve both reasonable efficiency per unit of compute hardware used and good speedup is vanishingly small.

The more important definition of a heterogeneous system is the one in which the component machines truly do offer a variety of special characteristics, especially clusters in which each machine may offer a different type of internal parallelism. In these cases, Java does not solve any of the key problems and non-Java code would need to be invoked to efficiently use the hardware parallelism. Use of Java threads can be seen as creating serious problems by forcing a non-transformable shared-memory MIMD execution model.

In summary, Java has many features that should be a part of a programming model for heterogeneous parallel computing. Unfortunately, 100% Java is not an appropriate model.