# Some Remarks on Parallel Processing and Java

*[email dated 6 Aug 1996]*

Geoffrey Fox

We can distinguish (at least) three forms of parallelism (concurrency) in Java of which the first two are reasonably uncontroversial.

a) Fine grain functional parallelism as exhibited by the built-in threads of Java. These could be very helpful in latency hiding by allowing several concurrent processes on a single node but do not naturally implement large scale parallelism.

b) Coarse grain functional or task parallelism or what the Linda group and Jim Browne would call coordination. This is roughly what is implemented in the Applet and network connection mechanisms of Java. This capability is the basis of WebFlow our proposed dataflow mechanism on the Web. Note that threads are shared memory but Applet mechanism is distributed memory parallelism.

c) Data parallelism is less clear for both technical and emotional reasons (Is it in the "spirit" of Java!). Let us discuss this in more detail.

In general, it seem plausible that data parallelism in Java should build on the corresponding discussions in FORTRAN and C++ (HPF and HPC++). Most relevant Java features are seen in one or both of these languages. In the following we list some considerations to be borne in mind in considering data parallelism in Java.

1. Data parallel FORTRAN or C++ typically compiles down to FORTRAN or C plus message passing. We note that the Java plus message passing (data parallel) model is uncontroversial. Thus there is no problem in defining the target implementation of a data parallel Java application. Further the Java equivalents of Fortran-M and C++ can be naturally defined.

2. The "Java plus message passing model" includes the case where "Java" immediately invokes a native class which could be an existing compiled C, Fortran or C++ and even an optimized Java code compiled directly for the native machine. Some argue that use of such (non-portable) native classes violates the philosophy of the Web or of Java. I disagree. I at least

download C code very often from the Web and current versions of Netscape illustrate how one is happy to download either the browser or plugins. We propose that users will be willing to download once and for all, a set of high performance Engineering and Science native classes. This implies that PCRC (Parallel Compiler Runtime Consortium) compiler runtime should be included in such a library and I expect this to be a critical part of any high performance Java environment.

3. The most powerful model assumes a WebServer (as opposed to a client) attached to each process in our "Java/Native Classes + Message Passing" model. This approach allows natural integration of Web computing as we have demonstrated in Kivanc Dincer's "HPF on the Web" prototype supporting Pablo performance and scientific result visualization from data passed by Java process to associated WebServer. Note that our standard "WebWindows" philosophy implies such a linked se of servers to coordinate computation.

4. Any efficient implementation must use "simple types" and not "objects" for distributed arrays as objects come with too much overhead. However we can make use of objects as a wrapper which stores at a high level overall information about array and links via intrinsic "methods" to the high performance native classes. Such a wrapper class does more than support data parallelism. It allows a general and convenient Java interface to existing C and Fortran data structures. This will allow easier development of Java based interfaces to existing simulations.

5. We suggest implementing data parallel Java using the HPF Interpreter approach we explored with Arpa funding and demonstrated (the work of Furmanski) in Supercomputing 93. The essential idea between the HPF Interpreter is simple. Take any Fortran90/HPF instruction such as:

   ```
   A = MATMUL(B,C)
   ```

   This can be executed in interpreted fashion without significant overhead as we are only concerned with cases that A B C are large arrays and time to interpret the single coarse grain array statement is small compared to its execution time even when interpreter invokes optimized parallel execution. Note that for MIMD parallelism, we imply large grain size in each process. Furmanski's HPF Interpreter was successful but we left it as a prototype as we did not have the resources necessary to complete a full blown system. Now Java and the Web have given us a more natural and powerful implementation and further our PCRC HPF infrastructure is much better.

6. We can implement the proposed data parallel Java as a main (host) class interpreting coarse grain statements linked to a set of child (native) distributed processes. This looks pictorially like:

```
Main (host)                 Interpreted HPJava statements
Java Class                  manipulating
                            Wrapper HPVector classes

Set of Child                Web Server running
(Native) distributed        a Java Interpreter invoking a
Processes                   Highly efficient ''node'' code
                            which is compiled Java, C and Fortran
                            using PCRC and MPI libraries etc.
```

7. We are suggesting this new `HPVector` class which is a data parallel array (and similarly for other parallel data structures). The `HPVector` class (of which `A`, `B`, `C` in 5 are instances) does not necessarily store array elements but rather user accesses elements through methods such as `A.grabelement(i1,i2)` to return `A(i1)` through `A(i2)`. We view `HPVector` class as a wrapper which links Java to an array in any relevant code including Java itself, F77, HPF, HPC++, F77 + Message Passing etc.

8. Wrapper `HPVector` methods will include `A.distribute()` and `A.align()` to implement HPF directives as calls to methods.

9. forall statements are very popular and powerful in HPF but are not so trivially implemented in our formalism as they involve array elements and not arrays. One possibility is to view a forall as implementing a new HPF array function in a flexible way and treat forall statement as a script which implements this new function. Thus something like:

```
forall(I=1 to 100)
        a(I)=b(I)*b(I+1)/c(I)
```

could be written as:

```
A=HPVector.forall("forall(I=1 to 100);
                   a(I)=b(I)*b(I+1)/c(I)",B,C);
```

10. The implementation of independent DO loops is also not so clear as really these reflect control and not data parallelism. Perhaps these should be implemented through task parallel (coordination) mechanism in Java.

11. Interesting features of this approach include the fact that no new language extensions are required (although you could add forall to language); it allows a (slow) pure Java sequential version as well as optimized parallel versions. It allows one to build both Java wrappers to existing applications and new parallel Java applications in the same formalism.

12. The main (host) class is naturally fully interpreted and the use of something like JavaScript (when it has been integrated with Java) is particularly natural.