The Role of APL and J in High-performance Computation[*]

Robert Bernecky
Snake Island Research Inc
18 Fifth Street, Ward's Island
Toronto, Ontario M5J 2B9
Canada
(416) 203-0854
bernecky@acm.org

Abstract

Although multicomputers are becoming feasible for solving large problems, they are difficult to program: Extraction of parallelism from scalar languages is possible, but limited. Parallelism in algorithm design is difficult for those who think in von Neumann terms. Portability of programs and programming skills can only be achieved by hiding the underlying machine architecture from the user, yet this may impact performance on a specific host.

APL, J, and other applicative array languages with adequately rich semantics can do much to solve these problems. The paper discusses the value of abstraction and semantic richness, performance issues, portability, potential degree of parallelism, data distribution, process creation, communication and synchronization, frequency of program faults, and clarity of expression. The BLAS are used as a basis for comparison with traditional supercomputing languages.

[*]This paper originally appeared in the ACM SIGAPL APL93 Conference Proceedings. [Ber93b]

1 Introduction

Single processor computers based on electrical technology are reaching their performance limits, due to factors such as fundamental limits of optical lithography and the speed of light. It is easy to envision silicon-based computers which are a hundred times faster than today's processors, but speed improvement factors of thousands are unlikely.

Therefore, in order to achieve performance improvements, computer architects are designing *multicomputers* – arrays of processors able to work concurrently on problems. Multi-computers are hard to program. In a computing world which is, by and large, still accustomed to the von Neumann single-processor computing paradigm, multicomputers present several problems:

- Most programming languages were designed from a von Neumann outlook, and inherently possess no more capability for parallel expression than does a cash register. Extracting parallelism from programs written in such languages is difficult, since when any parallelism inherent in the algorithm is discarded by the limited expressiveness of the language used to describe the algorithm. Wall's study of instruction-level parallelism [Wal91] obtained a median level of parallelism of 5, even assuming very ambitious hardware and software techniques were available.

- Algorithm design has, by and large, been done from the von Neumann viewpoint, blinding people to the potential for parallel solutions to problems. In the past 25 years, only user of APL and a few other languages have taken parallel algorithm design seriously.

- Matching algorithms to machine architectures is difficult; making portable algorithms which run well on a variety of network topologies is even harder. Most adaptations of scalar languages to parallel expression have been done from the standpoint of a particular machine design, and *require* that the application programmer explicitly embed those architectural assumptions in the application program. Those language designers have abdicated their responsibility to provide programming tools that can be effectively used by the masses. They have merely passed the problems of synchronization, communication, and data distribution on to the users, who must embed such architectural considerations in their programs. Such embedding inhibits program portability and thereby limits the utility of programs written in such dialects.

Given these and other such problems, what can we, as language designers and compiler writers, do to alleviate or eliminate them?

The thesis of this paper is that applicative array languages with adequate richness, such as APL and J, can do much to solve these problems. The following sections will deal with issues including the value of semantic richness and abstraction, performance issues, portability, potential degree of parallelism, data distribution, process creation, communication and synchronization, frequency of program faults, and clarity of expression. The BLAS [LHKK79] are used as a basis for comparison with traditional supercomputing languages.

2 A Brief Overview of APL and J

> Applied mathematics is concerned with the design and analysis of algorithms or *programs*. The systematic treatment of complex algorithms requires a suitable *programming language* for their description, and such a programming language should be concise, precise, consistent over a wide area of application, mnemonic, and economical of symbols; it should exhibit clearly the constraints on the sequence in which operations are performed; and it should permit the description of a process to be independent of the particular representation chosen for the data.
>
> Existing languages prove unsuitable for a variety of reasons. [Ive62]

Ken Iverson originally created APL as a notation for teaching mathematics, and it was only later that the idea of implementing APL as a language on a computer was seriously considered. That the design took place independent of any particular computer system is perhaps one reason why APL differs so greatly from traditional languages.

The primary characteristics of APL which set it apart from other languages are:

- Array orientation
- Adverbs and conjunctions
- Consistent syntax and semantics

APL is an array-oriented language – all primitives are defined by their actions on entire collections of data. This orientation provides inherent parallelism; its importance was only recently recognized by the designers of other languages [Cam89]. Scalar functions, which work on each element of their arguments independently, are a simple example of the power of APL. Consider the task of multiplying two tables of numbers, `x` and `y`, and adding a constant, `z`, to them. This can be written in APL as `z+x×y` and in J as `z+x*y`. For example,

```
       3+4 5 6×2 1 0
11 8 3
```

Note that the argument shapes are inherently known. There is no need for the programmer to maintain information about array properties independent of the array itself. Each array contains information as to its type, rank, shape, and value, which is automatically propagated from function to function.

APL and J are functional, in that arguments to verbs (functions) are entire arrays, called by value, and results are arrays. Side effects are rarely used or needed in APL.

APL includes the concept of adverbs and conjunctions (operators) that modify the behavior of verbs, just as they do in natural language. For example, the *insert* or *reduce* adverb, denoted `/`, *inserts* its left argument among the subarrays of its right argument. For example,

```
     +/1 2 3 4 ⍝ Sum
10
     ×/1 2 3 4 ⍝ Product
24
     ⌈/1 2 3 4 ⍝ Maximum
4
```

Adverbs are perhaps APL's most important contribution to computing. They provide a powerful, consistent way of describing many commonly required functions. Two of the most powerful adverbs in APL are inner product and rank. Inner product generalizes the matrix product of linear algebra to arrays of any dimension, and to any verbs, not just plus and times [Ber91a, Ber91b]. For example, they have utility in associative searching and computation of transitive closure (TC):

Inner product form	APL form	J form
Matrix product	`x+.×y`	`x +/.* y`
Associative search	`x∧.=y`	`x *./.= y`
TC step	`x∨.∧y`	`x +./.*. y`

The *reverse* verb (`|.`) reverses the order of the leading axis of its argument:

```
  tbl =. i. 2 3 4
  NB. 2 planes,3 rows,4 cols
  tbl NB. value of tbl
 0  1  2  3
 4  5  6  7
 8  9 10 11

12 13 14 15
16 17 18 19
20 21 22 23
  |. tbl NB. Reverse the planes.
12 13 14 15
16 17 18 19
20 21 22 23

 0  1  2  3
 4  5  6  7
 8  9 10 11
```

The *rank* adverb (`"`) [Ber87] specifies the dimensions of the arguments to which a specific verb is to be applied, and thereby greatly enhances the power of the verb. In this example of rank in conjunction with reverse it permits reversal of rows or columns as well as planes:

```
    NB. Reverse each plane
    |."2 tbl
  8  9 10 11
  4  5  6  7
  0  1  2  3

 20 21 22 23
 16 17 18 19
 12 13 14 15
    NB. Reverse each row
    |."1 tbl
  3  2  1  0
  7  6  5  4
 11 10  9  8

 15 14 13 12
 19 18 17 16
 23 22 21 20
```

Later examples will clarify rank can introduce parallelism in the form of SPMD (Single Program, Multiple Data) capabilities, and as a general way to obtain greater expressive power from even simple verbs such as addition.

The above overview is necessarily brief and incomplete. For a more complete view of the various dialects of APL and their capabilities, refer to texts on J [Ive96, Ive91] and APL [IBM94, BB93].

3 The Benefits of Abstraction

APL is more abstract than other computer languages, in terms of data storage methodology and the functional capabilities provided to its users.

3.1 Numerical Abstraction

APL differs from many other computer languages in that it deals with data in the abstract – numbers and characters, rather than ints, longs, reals, double precisions, and so on. Relational expressions produce Boolean results; adding to a Boolean produces an integer; dividing integers by an appropriate number results in reals, and taking the square root of negative numbers results in complex numbers. All this is done without the user's knowledge or permission, for better or worse. The problems with such an approach are well-known to the numerical analyst, who may prefer to have things screech to a halt when they are not firmly in control, or to be able to trap the event and take appropriate action.

However, there are advantages to expressing numbers as numbers, and letting the machine handle whatever conversions are required. Among those advantages are architectural independence and reduced code volume, allowing the programmer to concentrate on the problem at hand, rather than being concerned about how the computer is going to store numbers.[1]

By not *requiring* the user to specify the type of each array involved in a computation, the system is free to choose a representation which is most appropriate for the particular computer system and the verb currently being executed. For example, moving from 32-bit integer to 64-bit integer machines is transparent to an APL application. Moving from S/370 floating point format to IEEE format is also transparent for most applications, although numerical analysts and those who have stored character arrays as numbers for some peculiar reason are obviously going to be affected.

3.2 Abstraction of Relationals

The abstraction of treating relationals as verbs which return results, rather than embedding them in control

[1]There are ways to deduce data types, and otherwise ensure that they remain as you desire, but the need for them is the exception rather than the rule.

structures, increases their power and makes it possible to treat most control flow problems as data flow problems. For example, adding `5` to the numbers with a `3` residue of `2` in the array `eo` can be written in a loop-free and IF-free manner as:

```
eo+5*2=3|eo
```

This transformation from control structure dependence to data flow dependence is based on properties of arithmetic identities. Nonetheless, such transformations work well in a large number of cases, and J's gerunds handle the remainder.

Elimination of control structure dependencies is critical to performance because control structure dependencies often stall pipelines, whereas data dependencies need not. Although techniques are available for removing such dependencies from traditional programs [AKPW83], programs written using control flow techniques are often harder to read, understand, and maintain than the same program written without them. Compare and contrast the above expression on `eo` with the corresponding Fortran code using loops and an IF statement, or even with the Fortran 90 WHERE/ELSEWHERE construct.

3.3 Abstraction of Verbs

APL provides a large set of general array primitives, which are considerably more substantial than those provided by other languages. Data structure verbs include such facilities as tranposing, rotating, and reversing arrays of arbitrary type. Search primitives include grading, set membership, generalized array matching, and locate.

Making these verbs available as primitives has several beneficial effects on large scale computation:

- Productivity – by making frequently required facilities available as primitives, APL frees the programmer from the drudgery of having to write many lines of code to perform common functions such as searching for a part number. The primitive is there, ready for instant use.

- Improved code reliability – the primitives are written by professionals and are used daily in a myriad applications. There is no need to debug yet another hand-coded sort routine, or to discover that a system is running slowly because it uses a bubble sort.

- Portability and performance – an algorithm which is optimal for one machine architecture may perform dismally on another. For example, binary search may be wonderful on a Cray X-MP, but a Connection Machine can perform searching in unit time, and a binary search slows it down considerably. Abstraction of the required capability – set membership or sorting, for instance – leaves the system implementor free to write the best possible code for a specific platform, and the application programmer can rest assured that, in all but the most arcane cases, the system will do a better job than he or she can.

  Moreover, when an application is ported to another architecture, the user effort required to obtain that ultimate performance on the new architecture is zero. This is not the case with low level algorithms, which are highly dependent upon the peculiarities of today's architecture and today's technology. Todays's hot code may be tomorrow's dog – unrolled loops, which run quite well on most vector machines, may run slower on SIMD machines than does the original code. [2]

[2] Flynn's taxonomy of computer architectures includes *SISD*: Single Instruction, Single Data; *SIMD*: Single Instruction, Multiple Data; *MIMD*: Multiple Instruction, Multiple Data. A related acronym, *SPMD*: Single Program, Multiple Data, means

- New algorithms – advances in algorithm design have immediate payoffs at no cost to the application writer. In interpreted APL, for example, the orders of magnitude performance improvement which occur when someone installs a highly optimized inner product algorithm are immediately available to all users. In a compiled environment, merely recompiling the system will have the same effect. It is as if someone looked at all your programs, found all occurrences of inner product, and rewrote them for you.

  In a system with limited abstraction, where the user is forced to write at a low level, adoption of new algorithms, such as the Boyer-Moore string search, are available only by having each user take the time to redesign, recode, and retest each and every instance of such searches.

- Reduced code volume – just as mathematicians use a concise notation to describe their ideas in a uniform, easily read and verifiable manner, so does a concise notation benefit programmers. This has a large cost impact both in terms of reading programs, which is critical when maintaining or enhancing programs, and in the reliability of such programs. Since program failure rates are highly correlated with code volume, the bigger a system is, the more things there are that can cause it to fail. By reducing code volumes by an order of magnitude, APL and J improve code reliability and maintainability.

  Programmer code production rates are fairly fixed in terms of lines of code produced per day. Since a line of APL code does an order of magnitude more work than a line in another language, APL programmer productivity is much greater. This is why organizations such as brokerage house, where time is critical to profit, use APL extensively.

- Mediocre programmers [3] can write good code – those whose expertise lies in areas other than computing, such as chemistry, medicine, or engineering, are not likely to be interested in spending a large portion of their time learning the details of cache management, data distribution, code scheduling, or loop unrolling. They rightly consider the computer as a tool for obtaining an answer to a problem in their discipline, and often do not know if the method they use is the most elegant or efficient one available. As a result, it is common to see computer and supercomputer applications using tens or hundreds of times more resources than is, strictly speaking, necessary. This is due in part to naivity and part to the users' perception of how their own time is best spent. Most often, that time is spent in non-linear areas where a little bit of computer knowledge would help a lot – sorting, searching, and the like.

  By making highly tuned versions of such commonly required functions available to the user directly as primitives, APL helps users to write more efficient code. A computational chemist may not be able to write a binary search, but with APL search primitives, there is no need to do so. The straightforward and obvious solution using the primitive is also the optimal solution. Thus, mediocre programmers are able to produce code which both meets their needs and runs well.

the running of the same program on multiple processors.

[3]This is not intended as an insult. It is simply a recognition of the fact that not everyone can be, has the time to be, or wants to be, a computer wiz.

3.4 Data Structure Abstraction

Most programming languages expose the underlying data structure of arrays to the user. Some deem this a Good Thing, inasmuch as it lets the programmer use knowledge of that data structure to advantage. For example, in Fortran, a user can EQUIVALENCE two arrays, and deduce properties about array storage, including data type, adjacency of elements, and so on. In C, the use of *pointers* permits rapid access to array elements.

However, exposure of data structures is a two-edged sword. There are a number of Bad Things about exposure, which are neatly avoided by treating data structures as abstract entities:

- Aliasing – if an array can be referred to in different ways, then compilation of efficient vectorized or parallelized code to operate on that array can be difficult or impossible, because of the inability of the compiler to ascertain the absence of data dependencies among array elements.

  In the case of languages which support *pointers*, the problem is exacerbated, because, except in the most recent dialects, there is no firm knowledge at all about the contents of a pointer. This means that any pointer expression can potentially conflict with a reference to any variable.

- Array distribution – The order in which array elements are accessed impacts the performance of applications. Fortran stores column elements adjacently, which makes columnar access rapid, due to high cache hit ratios and storage interleaving. Access to adjacent row elements may be considerably slower, due to cache misses, storage bank conflicts, and page faults. In a multi-computer with distributed storage, access to an element may be affected by the network distance to the processor which holds the accessed element.

These problems can be dealt with most easily by hiding the physical attributes of array storage from the user, and by forbidding aliasing. Can this be done without significantly affecting performance? Let us find out.

3.4.1 Alias Avoidance

Aliasing occurs when an array may be known by more than one name. This occurs in Fortran with EQUIVALENCE, and in C with pointers. APL has no cognate to EQUIVALENCE. In a functional language, such constructs are undesirable – they make comprehension of a program difficult as well as making it hard to extend, maintain, and optimize programs which use such constructs.

Equivalencing as a storage management method is a harder call. Few languages have good support for non-rectangular data structures, except as application-controlled vectors of storage, or as recursive data structures, with their attendant overhead. Some such equivalencing may grow out of Fortran's static storage management. Lack of storage allocation tools, remedied but not automated in Fortran 90 [Cam89], is only part of the solution, because the designers of Fortran 90 dropped that task back onto the application writer as well [Ber91b, Ber91a].

APL, by contrast, has inherently automatic allocation and deallocation for storage. Arrays are created when required and deleted when no longer required. This eliminates many of the storage problems associated with Fortran. It probably does not, however, eliminate all of them.

Equivalencing has impact on vectorization and parallelization, in that data dependency analysis is further complicated by having what are really two or more arrays being manipulated as if they were the same array. The art of data dependency analysis has not yet reached the stage where all dependencies can be detected. In such cases, the compiler must take a conservative view, and refuse to vectorize or parallelize the offending code.

Functional programming is important in the avoidance of aliasing. Aliasing can only occur if objects are given names and synonyms. Function programming uses a number of techniques to avoid these situations. Among them are:

- Functional programming itself – by avoiding the use of side effects to alter global variables, function programming simplifies the task of data flow and data dependency analysis. This eases the task of vectorization and parallelization, since there are few or no dependencies among computations.

- Single assignment – this technique, used by languages such as SISAL [MSA+85], allows the use of named variables, but permits them to be assigned a value only once. Single assignment languages simplify the task of parallelization and vectorization considerably. Indeed, SISAL is regularly beating Fortran on a number of large numerical benchmarks [Can92], mostly because of this simplicity. However, SISAL and other such languages are perhaps not the best tools for end users, as their computational power is not complemented by a similar expressive power. APL has the potential to offer both expressive power and computational power, but has yet to prove itself on the latter in the supercomputing arena.

- Tacit definition – this notation, developed by Hui, Iverson, and McDonnell [HIM91, MI89], goes a step beyond functional programming. In functional programming, the unnamed results of computations are themselves permitted as arguments to other computations. In tacit definition, the arguments to the functions themselves are not explicitly named. For example, a definition of the arithmetic mean, written as:

  ```
  (sum x) dividedby shape x
  ```

  can be written explicitly in APL as `(+/x)÷⍴x` and in J as `(+/x)%#x`.

  J also provides a tacit form, in which *all* arguments are elided. The placement of arguments within an expression is determined solely by the presence of *forks* and *hooks*. Space does not permit a full discussion of tacit programming, but

  > A wide class of explicit definitions can be expressed in tacit form using the facilities of J [HIM91].

  The basic idea behind the fork is that if three verbs appear in isolation, they represent a *fork*, which is to be interpreted as follows: The fork `X (f g h) Y`, in which `X` and `Y` are values and `f`, `g`, and `h` are verbs, is:

  `(X f Y) g (X h Y)`

  Drawing the associated syntax tree for this expression immediately reveals why it is called a fork. The tacit program for the arithmetic mean can be written concisely in J using a fork as `(+/ % #)`.

  Tacit definition removes the complication of data flow and data dependency analysis from the compilation process. Second, it offers a degree of parallelism itself, in that even the simple fork presented above allows the computations using `f` and `h` to proceed in parallel.

3.4.2 Automatic Array Distribution

Storage allocation and distribution of data among processors for high performance is a critical and largely unsolved problem in the design of multi-computer systems. In any multi-computer system, arrays must be accessible by those processors using their elements.

In a shared storage system, access to non-local storage is often tens or hundreds of times slower than access to a processor's local storage. In such a system, failure to allocate array elements at a processor which accesses them frequently can negate the desired performance gain of using a multi-computer. Folk wisdom among multi-computer users is that *the data you want is always on the other processor.* How can we ensure that data is where it should be, when we want it to be there? Let us look at how it is done today.

The Steele status report on High Performance Fortran [Ste93] (HPF) offers some good background material on the issues dealt with in this paper. Two of the directives discussed in the report are the ALIGN and DISTRIBUTE statements. These are intended to ensure that data can be associated with specific processors, for maximum computational speed. Since library routines are often written for maximum performance with little concern for the caller's data, this will guarantee that each subroutine library routine will require a different data distribution!

Such directives, or *pragmas*, as they are more commonly referred to, are valid within the scope of Fortran 90, but one is led to wonder whether the need for such pragmas is caused by Fortran's semantic poverty. A semantically richer language offers more information to the compiler about what is going on, and much of the value of the pragma is rendered nugatory.

The assumption that subarrays should be associated directly with processors is shortsighted, and reflects today's architectural view of reality. Newer machines will have improved connectivity, and other concerns, such as the destination of the resulting data elements, may be more important than having the argument elements immediately close at hand.

A fundamental design principle violated by HPF is that of separation of the algorithm from its implementation – An implementation on a specific machine architecture is tangled up with today's engineering constraints, the state of the art of machine design, and so on. This produces a goulash, rather than an algorithm, resulting in code which is non-portable to machines whose design we can not yet envision.

APL hides the implementation from the user, and leaves the writer free to concentrate on the problem at hand. It is the responsibility of the compiler writer, not the user, to ensure that the application achieves peak performance on the target architecture. How can this be done?

Consider the task of array allocation. To avoid cache interference, and maximize the benefits of interleaved main store access, an array must be accessed stride-1 for maximum performance. Yet such access is highly dependent upon the target machine's architecture and configuration. Is the user to embed host-specific code around every loop, to spread arrays across storage in such a way as to achieve this performance level? Of course not. There is not enough time to do so, nor is it clear that the benefits of doing so are worth the application programmer's time to achieve it.

Suppose that the compiler was to undertake this task, and determine appropriate array storage methods for each array. It might, for example, append extra columns to an array, so that particular reference patterns would be optimal in an interleaved or multi-computer environment. It might broadcast multiple copies of array segments to different processors, based on knowledge of access patterns. This is difficult in a language which supports aliasing and EQUIVALENCE, because those constructs make strong assumptions about inter-element distance and storage allocation techniques.

In APL, on the other hand, all array accesses are abstract, and it is impossible for a user to discern the internal storage representation used for an array. Therefore, there are no inhibitions to performing such storage management optimizations, and it can all happen with no work by the application writer. This provides another benefit, in that programs written without machine dependencies are inherently portable.

Abstraction makes a language easier to teach, to learn, and to use. If you are not concerned with the details of the underlying machine's architecture, then you can concentrate on your problem and its solution.

4 Expression and the Programmer

A semantically rich language is of immense value because knowledge in one area benefits another – learning ten verbs and twenty adverbs gives the potential for specifying 200 different actions. Similarly, a computer language with conjunctions and adverbs offers richness of expression to the programmer. For example, ISO Standard APL permits any of the dyadic verbs in Figure 1 to be used in conjunction with the *reduce* or *insert* adverb, in a consistent manner. Insert places the verb between the subarrays of the right argument, then evaluates the resulting expression. Thus, summation is expressed as `+/`, alternating sum is `−/`, product is `×/`, maximum is `⌈/`, and so on. Modern APL dialects permit any verb, including user-defined verbs, to appear as the left argument to insert. For example, `+.×/` can be used to multiply a chain of matrices together.

The simple and consistent behavior of adverbs and conjunctions in APL gives the programmer an excellent set of parts from which to construct the specific tool required to solve a particular problem. This erector set approach is in sharp contrast to the Swiss army knife approach taken by Fortran 90, in which the set of tools is limited by the imagination of the language designer, rather than by the imagination of the user. For example, there is no alternating sum reduction in Fortran 90, although such sums occur frequently in physics and engineering.

The value of richness of expression goes beyond its utility to the programmer. It makes life easier for interpreter and compiler writers, who can exploit a single advance over a wide range of areas. As a real-life example of this, consider the inner product conjunction in APL.

In APL, the inner product `f.g` denotes a matrix product in which `g` is the function used to combine argument array elements, and `f` is the function used in the reduction into the result. The traditional matrix product of linear algebra is written as `+.×`; one step of a Boolean transitive closure on an adjacency matrix is `∨.∧`.

In the late 1970's, I was manager of the APL development department at I.P. Sharp Associates Limited. A number of users of our system were concerned about the performance of the `∨.∧` inner product on large Boolean arrays in graph computations. I realized that a permuted loop order would permit vectorization of the Boolean calculations, even on a non-vector machine.[4] David Allen implemented the algorithm and obtained a thousand-fold speedup factor on the problem. This made *all* Boolean matrix products immediately practical in APL, and our user (and many others) went away very happy.

What made things even better was that the work had benefit for all inner products, not just the Boolean ones. The standard `+.×` now ran 2.5 – 3 times faster than Fortran. The cost of inner products which required type conversion of the left argument

[4]This algorithm was an outgrowth of an early non-Boolean algorithm used in CDC STAR-100 APL, probably due to Mike Grimm.

ran considerably faster, because those elements were only fetched once, rather than N times. All array accesses were now stride one, which improved cache hit ratios, and so on. So, rather than merely speeding up one library subroutine, we sped up a whole family of hundreds of such routines (even those that had never been used yet!), with no more effort than would have been required for one.

Similarly, supercomputer techniques, such as block algorithms, for speeding up matrix product are, by and large, applicable to a much wider range of problems than generally thought. Use of APL makes high-performance solutions to those problems immediately accessible to all users.

As noted elsewhere, the algorithms underlying such critical functions are invariably tuned to a specific host. The benefits of keeping the algorithms and their tweaking out of the application should be obvious. Portability and performance suffer. Indeed, this was one of the justifications for the creation of the Basic Linear Algebra Subprograms (BLAS):

> ...general agreement on standard names and parameter lists for some of these basic operations ...would add the additional benefit of *portability* with *efficiency* ...[LHKK79]

The same article also presents a cogent argument for use of APL, although it is aimed at promoting the BLAS:

> It can serve as a conceptual aid ...to regard an operation such as the dot product as a basic building block... It improves the self-documenting quality of code to identify an operation such as the dot product by a unique mnemonic name.

5 Performance and the BLAS

The Basic Linear Algebra Subprograms (BLAS) are a set of Fortran-callable subroutines which perform operations which are commonly required in high-performance computation. They are typically highly tuned to specific architectures in order to obtain maximum performance on each host.

BLAS are categorized according to their computational complexity. Some of the level-3 BLAS and the APL expressions which correspond to their definitions are [DDHD90]:

$C \leftarrow \alpha AB + \beta C$	C←(α×A+.×B)+β×C
$C \leftarrow \alpha A^T B + \beta C$	C←(α×(⍉A)+.×B)+β×C
$C \leftarrow \alpha AB^T + \beta C$	C←(α×A+.×⍉B)+β×C
$C \leftarrow \alpha A^T B^T + \beta C$	C←(α×(⍉A)+.×⍉B)+β×C

Other BLAS are harder to describe as simple APL expressions, because they involve operations on symmetric or triangular arrays. Such BLAS are properly considered as applications, since they make assumptions about characteristics of arrays which are not a part of most computer languages.

However, even with such limitations, techniques such as *array morphology* [Ber93a], offer the potential for discovering and propagating such information in APL. Array morphology is the study of the array properties in array-based languages. Characteristics of arrays may be deduced from algebraic identities and properties, such as that the sum of an array and its transpose is symmetric. The semantic level of APL is high enough that detection of $B^T + B$ is easy. APL interpreters often use pattern matching techniques to find such phrases in APL, often called *idioms*, and interpret them with code tuned to handle such special cases efficiently. Perlis offers a number of insightful examples of what he calls mini-operations [Per79], many of which are detected by APL interpreters. His brief paper is well worth reading as an introduction to the power of the language.

Assertions are another way to specify array characteristics [Ber93a]. Asserting that an array is symmetric could allow a compiler to produce appropriate code for its manipulation. For example, the transpose of such an array would be treated as an identity, and would generate no code.

Even with such techniques, I doubt if raw APL will be able to compete favorably with all of the BLAS, which are traditionally hand-coded to squeeze the last bit of performance from a system, just as raw Fortran cannot compete favorably with the BLAS. Many of the BLAS appear to be applications more than they are subprograms, and their complexity cannot be dealt with by such trivial expressions, even in APL. Of course, an idiom recognizer could easily interface to the BLAS, but that's not the point.

APL can compete favorably with the BLAS when we run into the Procrustean nature of the BLAS – if your application does not fit the BLAS definition precisely, you are out of luck. For example, when applied to complex numbers $x_j + iy_j$, the *SASUM* function computes $\mid x_j \mid + \mid y_j \mid$ instead of $(x_j^2 + y_j^2)^{1/2}$. Having a high-performance version of a function you can not use is of little value. By offering excellent, but perhaps not quite ultimate, performance on such simple expressions, APL can meet the needs of the majority of users who need something a bit out of the ordinary.

Finally, the BLAS leave the bulk of the effort in porting to different architectures to the user. In a discussion of block updates, Dayeè and Duff [DD90] suggest the use of

> ... JIK-SDOT for short vector length architectures ... *and* KJI-SAXPY for all other cases.

What this means is that the user has to tinker with the application when porting it to a machine with a different architecture. This is akin to swapping the positions of the accelerator and brake when moving from a 4-cylinder automobile to a V-8. The redesigned inner product algorithms discussed earlier dynamically pick an algorithm based on the relative sizes of the arguments, their types, available storage, and several other parameters. Surely, such choices should be made automatically and should not devolve to the user.

6 The Potential for Parallelism

The most important aspect of APL and J as they relate to large scale computation is the amount of parallel computation which is inherent in the notation. Both APL and J possess considerable parallelism at a number of semantic levels, including:

- Primitive verbs
- Adverbs
- Expression
- Phrasal forms (J)
- Defined functions
- Cells and frames (J and certain APL dialects)
- Composition (J and certain APL dialects)
- Tacit definition (J)
- Gerunds

Parallelism can be exploited concurrently at all of these levels. As will be shown in a later section, synchronization and communication among parallel processes is largely inherent, and the programmer can avoid thinking about those problems. The following section briefly discusses the parallel properties of APL and J. It expands on a recent article by

Willhoft [Wil91], which limits itself to APL2 and does not discuss expressions or verb trains.

A detailed study of all of the capabilities of APL and J is beyond the scope of this paper. The following sections will merely offer a few examples from each class where parallelism can be exploited.

6.1 Primitive verbs

APL possesses a rich set of primitive verbs (functions) which are inherently parallel in nature. They may be categorized by the shape of the arrays to which they naturally apply, as rank-0, rank-1, rank-2, and so on.

6.1.1 Scalar or rank-0 Verbs

Rank-0 verbs, the so-called *scalar functions*, obtain their name from their characteristic of independent operation on each scalar element of their argument array(s). Figure 1 shows the operations which are in both APL and J as scalar verbs. Fortran 90 has included some of these scalar verbs in its *numeric functions* and *mathematical functions*.

A scalar verb applies independently to each element of its argument. That is, there is no communication required among the elements. For example:

```
        1 2 3<4 2 0
1 0 0
```

Thus, all scalar verbs represent instances of fine-grain parallelism, which means they are simple to implement efficiently on vector or parallel machines. Expressions consisting of these verbs can exploit the *chaining* capabilities of vector architectures and data distribution on SIMD or MIMD machines.

Symbol		Meaning	
APL	J	Monadic	Dyadic
+	+	Conjugate	Add
−	-	Negate	Subtract
×	*	Signum	Multiply
÷	%	1 Divided by	Divide
⋆	^	e^y	x^y
⍟	^.	Base *e* log y	Base x log y
⌊	<.	Floor	Minimum
⌈	>.	Ceiling	Maximum
<	<		Less than
≤	<:		Less or equal
=	=		Equal
≥	>:		Greater or equal
>	>		Greater than
∨	+.		Logical or
∧	*.		Logical and
⍱	+:		Logical nor
⍲	*:		Logical nand
∼	~	Logical not	
○	o.	π×y	sin, cos, etc.
?	?	Roll	Deal
|	|	Absolute value	Modulus

Figure 1: Scalar verbs in APL and J

6.1.2 Non-scalar verbs

Non-scalar verbs are primarily used for selection, structuring, and searching operations. By definition, they are defined on array structures larger than scalars, so it is natural to seek parallelism in them. Since space prohibits a full analysis of them, only two examples are presented. Willhoft offers a fuller discussion of the topic.

APL contains two sort-related verbs, *upgrade* and *downgrade*. They return a permutation vector for the argument which, if used to index subarrays, will bring the argument array into increasing or decreasing sorted order, respectively. Since the literature on parallel sorting [Lei92, Sto87] is extensive, it will not be discussed here, other than to note that the effort required to parallelize or vectorize a sort in APL is trivial, because all the required information is immediately at hand.

A simple example of the power of APL is exemplified by two versions of a convolution verb,[5] due to George Moeckel of Mobil Research. The first uses the non-scalar *rotate* verb (⌽) to skew the result of the outer product before reduction. The second uses rotate to generate skewed versions of the filter/wavelet before performing an inner product with the trace:

```
r←wz conv tr;npad;h
h←wz∘.×tr,(npad←(⍴wz)-1)⍴0
r←(⍴tr)↑+⌿(0,-⍳npad)⌽h

r←wz convo tr;npad;h;n
h←tr,(npad←(n←⍴wz)-1)⍴0
r←(⍴tr)↑wz+.×(0,-⍳npad)⌽(n,⍴h)⍴h
```

[5]Bob Smith, now of Qualitas,Inc, designed a convolution conjunction for APL [Smi81], noting that its "applications include polynomial multiplication, substring searching, and weighted moving averages." A well-implemented version of such a conjunction would be a worthy addition to J and APL.

With the use of the rank adverb, these verbs can be simplified. For example, the expression:

`(0,-⍳npad)⌽(n,⍴h)⍴h`

may be written as `(0,-⍳npad)⌽⍤0 1 h`.

That is, use each scalar on the left (the integers `0,1,2. . . n-1`) as the rotation amount for the filter. Since there is only one filter, it is reused and the result is a table of `n` rotated filters.

In a naively implemented environment, this use of rank will reduce processor and storage requirements for the skewed filter by half. In a more sophisticated environment using dragalong [Abr70], the generation of the skewed filter could be avoided entirely.

The potential for parallelism here is considerable. There is no interplay via side effects among the arrays, and the computation neatly decomposes into a separate computation for each result element.

6.2 Rank, Cells, and frames

The concept of function rank is fundamental to array parallelism in APL. The *rank* of a verb specifies the number of axes in the arrays to which the verb naturally applies. For example, addition (`x+y`) is defined on scalars adding to scalars, so is rank 0 0. Matrix inverse (`%.y`) is defined on tables (matrices), so is rank 2. Rotate (`x|.y`), or end-around shift, is defined on a scalar left argument, which specifies the number of positions to be rotated, and a list or vector right argument to be rotated, so rotate is rank 0 1. Since the ravel verb makes its entire argument into a list, it is classified as rank ∞.

Operation of a rank-k verb upon arrays of higher rank than k is defined as *independent* application of the verb to each rank-k subarray of the argument, with the final result being formed by laminating the individual results. There is no specified temporal ordering, so side effects can not be depended on. Data dependencies simply do not exist.

Consider a few examples of how this works in practice on an array `x` whose shape (`$x`) is `2 3 5 4`. Matrix inverse applies independently to each of the six (2 by 3) cells of shape `5 4` to produce a result of shape `2 3 4 5`:

```
        $%.x
2 3 4 5
```

That is, each inverse produces a result of shape `4 5`, and the six of them are laminated into the 2 by 3 frame.

The determinant produces a rank-0 scalar for each result from a rank-2 argument, so their lamination produces a result array which is of shape `2 3`, because the scalars do not contribute to the result shape.

Fortran 90 has the same effect as APL on scalar functions, but because Fortran 90's semantics are not generalized to the entire language, its applicability is limited. It also complicates the semantics of the remainder of the language.

Extension occurs when one argument to a dyadic verb is of the same rank or less than the defined rank of the verb. That argument is extended by reusing it as many times as needed. For example, the expression `2 2 2+2 3 4` and the extended expression `2+2 3 4` both produce `4 5 6`.

Extension has even greater power when combined with adverbs or conjunctions, as will be shown later.

The concept of function rank has considerable value. It is:

- a conceptual tool to guide our thinking about algorithms,
- a design framework to assist in language design decisions, and
- a powerful way to express SPMD parallelism in a concise and uniform fashion.

Although all verbs have a defined rank, it is often convenient to alter that rank to meet the needs of specific algorithms. This is done with the rank adverb, to be discussed shortly.

6.3 Adverbs and Conjunctions

As noted earlier, adverbs and conjunctions are a central factor in APL's power as an algorithmic notation. They allow one to concentrate on the problem at hand without getting bogged down in detail. In a sense, they are macros which allow the user to fill in the blanks for a specific computation. For example, in the inner product, the user specifies the combining and reducing verbs, while the control structure used remains unchanged.

6.3.1 Insertion and Scan

In *reduce* or *insert*, the user specifies a verb to be inserted among the subarrays of the argument, and then evaluated. This produces a wide range of useful functions, including the summation, product, maximum, minimum, all, any, and count of Fortran 90, as well many that are not present in Fortran 90.

The *scan* adverb, often called a *parallel prefix* operation, has immense power, particularly on numeric and Boolean arguments. Scan is defined as producing the partial reductions on an argument. For example, the sum scan produces:

```
      tbl
2 3 4    5
1 1 1    1
9 2 0.5 10
      +\tbl
2  5  9   14
1  2  3    4
9 11 11.5 21.5
```

Boolean scans have such significant value that a number of articles have been written on them alone. For example, not-equal scan can be used to locate quoted strings in text (with the help of the = verb to locate the quotes and generate the required Boolean).

6.3.2 The Recurrence Relation

Recurrence relations are often trivially expressible in terms of scan. A recurrence relation is a relation among the elements of a list or vector such that the following holds:

```
seq[i] = add[i]+mpy[i]*seq[i-1]
```

Until recently, the recurrence relation was generally considered "non-vectorizable" by compiler writers for vector machines. Back in 1971 or earlier, APLers had not been told this was impossible, so they began writing the recurrence relation as:

```
t×+\add÷t←×\mpy
```

John Heckman created user-defined functions to perform a number of scans in APL, which did the job in $\log_2 n$ iterations. His algorithm was extremely fast and worked on arrays as it does on lists, so many calculations of mortgage payments, etc., could be made at once. The potential for parallelism was obvious. The scan adverb entered APL as a primitive in 1973.

6.3.3 The Rank Adverb

The rank adverb ("k) permits customization of a primitive, derived, or defined verb to operate on arrays of rank k. Consider the humble +. Operating in isolation, it can only add array to array or scalar to array. But working in conjunction with rank, it can do much more. Here it is used to add a list to each row of a matrix:

```
      100 200 300+"(1) 2 3$0 1 2 3 4 5
100 201 302
103 204 305
```

Rank is applying addition to each rank-1 subarray of both arguments. Since there is only one left argument (100 200 300), it is extended and added to both rows of the right argument.

```
       100 200+"(0 1) 2 3$0 1 2 3 4 5
100 101 102
203 204 205
```

This picks rank-0 elements from the left argument, and rank-1 subarrays from the right argument. In this case, there are two cells in the left argument, and two in the right, so no extension occurs at the rank level. However, extension does occur at the level of the scalar verb, when 100 is added to 0 1 2.

The use of upgrade with rank has some interesting applications. For example, consider the task of determining which words in a table are anagrams of one another, a problem posed by Jon Bentley [Ben83a, Ben83b]. This can be done in J or APL in a few characters, with no explicit iteration [Ber87]. The key to an elegant solution is combining a sorting verb, upgrade, with rank, to grade each name independently. In J, this is done as /:"1. This results in as many sort operations as there are words in the list, and each of those sorts may be itself parallelized. To be fair, the amount of parallelism obtainable in sorting the characters in a single word is small, but the point is that there is significant parallelism even in a 4-character expression.

6.4 Expressions

For reasons of comprehension, maintenance, and simplicity, APL programs are usually written as applicative functions. It is, therefore, enlightening to view them from that perspective in order to determine what level of parallelism and other high-performance computing benefits might exist within them.

The following discussion assumes, for simplicity, that embedded assignment in mid-expression is not used. Although treatment of such assignments only slightly complicates the analysis, it is beyond the scope of this introduction to the topic.

Consider that expressions are formed by combining nouns (`1 2 3.5 4j3`), pronouns (`X, foo, dx`), verbs (`+`,`-`,`*`,`#`), and adverbs and conjunctions (`/, \, ", .`) in syntactically and semantically meaningful ways. For example:

```
((a+b)*d)%(|:foo e)-f^g
```

That is, the sum of `a` and `b` is multiplied by `d`, that result is divided by the result of the transpose of `foo` applied to `e` minus `f` raised to the `g` power. The linear progression of the computation does not immediately suggest that any parallelism is present.

In fact, there are two types of parallel behavior within the expression which may be exploited concurrently or separately:

- *parenthetical parallelism*
- *chaining* of partial results, or *loop jamming*.

In expressions like the above example, it is obvious by inspection that computation of all of the innermost parenthesized expressions can proceed in parallel. That is, the computation of `(a+b)` can occur at the same time that `(|: foo e)` is being computed. In practice, care must be taken to ensure that side effects within called functions, such as `foo`, or mid-expression assignment [6] do not alter the semantics of the expression. Since we assume that functional programming style is being observed, this will not happen.

Since these parenthesized computations typically operate on entire arrays, and since `foo` may represent an large amount of computation, itself amenable to all the forms of parallelism presented here, there may be substantial benefit to seeking parallelism at the expression level.

Furthermore, if the expression is fully parenthesized to reveal its order of evaluation, even more parallelism becomes evident:

```
(((a+b)*d)%((|:foo e)-(f^g)))
```

It is now clear that the computation of `(f^g)` can also occur in parallel. Once these innermost parenthesized expressions complete, producing temporary array results, we are left with:

```
(T1*d)%(T2-T3)
```

Repeating the parallelize-the-innermost-expressions process, we see that `(T1*d)` can proceed in parallel with `(T2-T3)`.

It is now fairly obvious that, at any stage in the evaluation of an expression, there are as many parallel execution threads as there are innermost expressions in the fully parenthesized expression.

Chaining, or loop jamming, offers the potential for parallelism over a different axis. In scalar languages, the computation of `((a+b)*d)` would probably be written as:

```
do i=1,n
T1(i)= (a(i)+b(i))*d(i)
enddo
```

rather than as:

```
do i=1,n
T1(i)= a(i)+b(i)
enddo
do j=1,n
T2(j)= T1(j)*d(j)
enddo
```

[6] Hence its omission in this brief treatment.

Although code sequences such as the latter are rarely written directly, they do arise out of program-generated code and out of compiler-induced optimizations. Therefore, compilers routinely perform loop jamming to reduce them to a form such as:

```
do i=1,n
TMP= a(i)+b(i)
T1(i)= TMP*d(i)
enddo
```

A similar form is also used by some vector supercomputers, notably the Cray X-MP, to *chain* together vector operations, so as to overlap their computation across several functional units, and to reduce the bottleneck of traffic between main store and the registers where computation actually occurs.

Static analysis of APL functions permits similar optimizations to take place at the expression level, merging a sequence of primitive functions on arrays into an interleaved execution on subsets of the arrays. This merging makes the expressions amenable to optimized execution by techniques such as vectorization, parallelization, and strip mining. IBM's APL2 interpreter for the IBM 3090 Vector Facility performs loop merging, although the documentation [MM89] is vague about the extent to which it is actually done.

Loop jamming has the potential for significant performance in an interpretive environment, because it reduces the load/store traffic associated with array-valued temps, as well as eliminating the storage management overhead associated with each jammed primitive. This has the desirable effect of reducing $N_{1/2}$ for the interpreter. [HP90] [7] Large $N_{1/2}$ values are the well-founded basis for interpreted APL's reputation of being slow for iterative computation on scalars and small arrays.

[7]$N_{1/2}$ is a measure of the minimum number of elements required in an array computation for the computer to achieve half of its peak performance on that computation. Since $N_{1/2}$ is effectively a measure of the "0 – 60 time" for a system, the smaller it is, the better.

6.5 Phrasal forms

Phrasal forms include the fork and hook discussed previously. Since their utility from the standpoint of parallelism is obvious, they are not discussed further here. More information on phrasal forms can be found in McDonnell and Iverson [MI89] and in McIntyre [McI91].

6.6 User-defined Functions

User-defined functions are APL's cognate of functions and subroutines in other languages, and are the primary method used to create non-trivial applications in APL. Like other languages, control flow in user-defined functions is sequential and appears, in isolation, to be inherently non-parallel. Of course, since user-defined functions may utilize all the forms of expression discussed in this section, there is considerable potential for parallelism within each sentence of APL. In spite of the apparent plodding nature of sequential control flow, there are two ways in which APL functions may exhibit parallel behavior:

- SPMD computation at the function level
- concurrent inter-line computation

SPMD computation occurs when the function is invoked multiple times by an adverbial expression. The two most common forms of this invocation are through the *rank* adverb and through the *under* conjunction, of which the APL2 *each* adverb is a special case.

Consider a function `p` that solves Poisson's equation using any of several techniques, such as Jacobi or Gauss-Seidel iteration on a 2-d grid. How can this be applied to several independent sets of data? Three ways come to mind:

- Modify the function to accept a rank 3, rather than rank 2, argument, and make it explicitly handle multiple sets: `p d3`
- Use the rank adverb (`"k`) to explicitly apply the function to the rank 2 tables of a higher rank argument: `p"2 d3`
- Use the *under* conjunction (`&.`) to apply the function to each of an array of tables: `p&.> d2a;d2b;d2c`

Each of these techniques has its own set of advantages and disadvantages. The first one obviously requires a perhaps non-trivial amount of programming effort. Both the first and the second require all sets of data to be the same shape. The third may be, with current interpreters, slower or more of a storage hog than the first two approaches. Thus, choice of a method depends in part on the set of problems being solved, and partly on taste.

The first may or may not exhibit more parallelism than the last two, depending on how the code is written. If it were to iterate over all sets until all had stabilized, then toward the end, it would be wasting effort on the sets which had already stabilized. On a machine with a large amount of fine-grain parallelism, this might not make much difference. The latter two are more appropriate for large-grain parallelism, since they decompose the problem into a series of logically independent, fairly large computations, which may all proceed concurrently.

Inter-line parallelism is essentially the same form of parallelism which exists at the expression level, except that the analysis extends across sentences. Unlike the naive form of expression level parallelism discussed above, this form must take assignment and other side effects into account, if semantics are to be preserved properly. Since the essentials of inter-line and expression level parallelism are much the same, no further discussion of it will take place here.

6.7 Composition

Composition in J and the SHARP APL dialect of APL offers significant computational power [Ber87, BI80]. Unlike composition in mathematics, which does little more than to glue functions together, composition in APL also glues together the intermediate results. APL requires that the result of any computation be a rectangular array. There are ways of getting around this, by use of recursive data structures or indirection, but they are sometimes inconvenient or messy.

Consider composition in J, denoted as `&`, combining two verbs, `f` and `g`. The composition applies `g` to each cell of the argument(s) as determined by `g`. The result of each cellular computation from `g` is then passed to `f`, without the requirement that each result from `g` be of the same shape. Effectively, such composition lets us *pipeline* arrays from verb to verb, offering considerable convenience as well as a great deal of MIMD parallelism: Each cell can be computed independently in any of the composed verbs. Unlike the pipes of operating systems, which only support a single character vector, these pipes are arrays of arbitrary type, rank, and shape.

6.8 Tacit definition

Tacit definition is an extension of phrasal forms and functional programming, in which variable names do not appear [HIM91]. Hence, issues of liveness of variables do not arise, and the compilation task is simplified in that regard.

Donald McIntyre's delightful history of mathematical notation includes a presentation of eight statistical functions in traditional and tacit form [McI91].

6.9 Gerunds

In natural language, a gerund is the noun form of a verb. For example, in the phrase, *programming is an art*, the verb *to program* is a gerund. In J, a gerund is an array which represents one or more verbs. Gerunds are manipulated identically to any other data in J. For example, IF/THEN/ELSE can be interpreted as using the Boolean value resulting from the conditional to index a two-element gerund, and then applying the selected element to an appropriate argument. Gerunds were introduced into J as a way to offer a rich set of capabilities including:

- MIMD computation
- control structures including generalized if/then/else, case, do while, etc.
- first-class treatment of verbs

More information on gerunds may be found in Bernecky and Hui [BH91] and in Iverson[Ive96, Ive91]. Background material may be found in Bernecky's early work on function arrays[Ber84].

First-classness is not of great relevance to this discussion, except inasmuch as it enables clean design of other capabilities, by supporting computation on functions as data. For example, arrays of verbs may be created, manipulated as if they were data objects, then a subset of them applied to data objects.

Control structures are important to high-performance computing because they simplify control flow analysis and data flow analysis. This in turn permits generation of more efficient code and enables vectorization and parallelization:

> Loops without dependencies among their iterations are a rich source of parallelism in scientific code [HSF92].

Flow analysis in APL is even more important than in other languages because APL does not have declarations. Methods such as array morphology are only now beginning to be used to optimize APL execution. Gerunds make APL more amenable to flow analysis and simplify the generation of high quality, efficient code.

Gerunds also offer the programmer a direct and simple way to specify MIMD (Multiple-Instruction, Multiple-Data) computations. There are several ways to achieve MIMD, including agenda and insertion of gerunds.

6.10 Parallelism Measurement Criteria

I have considered ways to measure the potential parallelism of an entire APL application, but thus far, have not come up with a simple, satisfying metric which is not a function of an arbitrary number of variables for any but the most trivial operations. The ability to nest parallel structures within APL makes the description of a program's parallelism a tree structure rather than a scalar.

7 Processes and Synchronization

As was shown previously, APL has the capability to describe parallel computations on a variety of levels without resort to processes, operating system characteristics, process creation, synchronization, process destruction, etc.

Although these nuts-and-bolts aspects of parallel computing on today's machines may or may not be with us tomorrow, they certainly should not be embedded in the description of algorithms.

The inherent parallelism of APL handles all of the above requirements implicitly. It offers an embarrassment of riches from the standpoint of parallelism. Unlike other languages, the problem in APL is *not* determining where parallelism exists. Rather, it is to decide what to do with all of it.

8 Portability

As noted throughout this paper, the abstract nature of a language such as APL has a significant impact on the portability of applications created with it. Not only are programs more portable than they would be if assumptions about underlying data types and representations were made, but the performance potential of applications, particularly across wide architectural boundaries, remains higher with abstract languages than it does with low-level languages such as Fortran.

9 Bugs

Notational consistency is an important aspect of APL dialects. The language has a simple syntax, and consistent semantics for both primitive and user-created entities. This means that issues such as precedence, a frequent cause of bugs in traditional languages, are simply not an issue in APL.

As noted previously, the conciseness of APL also contributes heavily to code reliability, as does the inherent preservation of type and shape information with arrays. For example, in the APL model of a loom [Ber86], there is one parameter – the tieup matrix used to connect the foot treadles to the harness – and two arguments: the threading, which specifies which warp thread is connected to which harness, and the treadling sequence actually used. The APL code is entirely functional and clearly correct, consisting of two index operations and an inner product. Contrast this with the several pages of BASIC to perform the same task, in which there is so much baggage that it is a challenge just to find the part of the code which actually does the work!

APL's inherent array-handling properties and functional nature contribute to the reduction in explicitly coded variables representing array size, induction variables, and so on. With less code and fewer variables, fewer things can go wrong, producing a net increase in code reliability.

Although these aspects of APL do not heavily correlate with supercomputer usage, they are nonetheless noteworthy as being features of the language which help one to obtain correct answers from a computer in a shorter timeframe than is otherwise possible.

10 Clarity of Expression

The clarity of expression of ideas possible with APL over that possible with languages such as Fortran makes it easier to optimize algorithms and to gain insight into problems. The performance of an APL model of a loom was improved by a factor of twenty in a short time, by virtue of the clarity of the algorithm expressed in APL – the APL program consisted of three verbs, whereas the same program written in BASIC was more than two pages of code.

Conciseness works well with our short term memory – we can only deal with seven or eight symbols or chunks of information at once, and APL, by condensing expression into a compact form, allows us to grasp a larger portion of the problem at once.

This conciseness also has benefits for compilers, because APL provides a larger amount of semantic context from which to deduce properties of an algorithm, and thereby produce improved, parallel, or vectorized code.

The abstraction of APL makes it possible and desirable to isolate architectural characteristics from algorithms. This enhances portability, and increases the clarity of the algorithm, by not cluttering it up with details about the machine on which it happens to be running today.

11 Summary

APL has been shown to possess an immense amount of parallel expression, providing a rich blend of SIMD, MIMD, and SPMD capabilities. It does this without compromise – there is no need to embed machine characteristics within application programs. This enhances portability.

APL's richness of expression provides significant semantic content for the compiler writer, which makes generation of high-performance code an easy task.

12 Additional Reading

Cann [Can92] offers a number of the same arguments presented here, as well as concrete evidence, in terms of benchmarks on non-trivial numerical programs, that applicative languages have the potential to match or beat the performance of Fortran.

Willhoft's article covers some of the same ground as does this article. He concentrates on the semantics of each primitive, to present its parallel nature. He does not discuss parallelism at the expression or function level, nor does he discuss parallelism of the sort expressible by phrasal forms or gerunds (APL2 does not possess gerunds). Willhoft makes pragmatic measurements of the potential parallelism present in a small number of APL applications, drawing conclusions which are in line with those presented here.

Some of Willhoft's recommendations for language changes already exist in J, in the form of gerunds to provide control structures and in the rank adverb as an axis specifier. Sadly, parallelism in the APL2 *each* adverb is said to be unachievable in APL2 because it lacks the ability to specify that a user-defined verb is free of side effects. This is a design issue, which has been resolved in other systems by defining the behavior of the each adverb or its cognate as having undefined application order on its argument.

13 Acknowledgments

Professor Ken Jackson of the University of Toronto Computer Science Department encouraged me to produce this paper. I am grateful to Diane V. Flores for her editorial assistance.

References

[Abr70] Philip Abrams. *An APL Machine*. PhD thesis, Stanford University, 1970. SLAC Report No. 114.

[AKPW83] J.R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the Principles of Programming Languages Conference*, 1983.

[BB93] Robert Bernecky and Paul Berry. *SHARP APL Reference Manual*. Iverson Software Inc., 33 Major St., Toronto, Canada, 2nd edition, 1993.

[Ben83a] Jon Bentley. Programming pearls. *Communications of the ACM*, 20(8), July 1983.

[Ben83b] Jon Bentley. Programming pearls. *Communications of the ACM*, 20(9), August 1983.

[Ber84] Robert Bernecky. Function arrays. *ACM SIGAPL Quote Quad*, 14(4):53–56, June 1984.

[Ber86] Robert Bernecky. APL: A prototyping language. *ACM SIGAPL Quote Quad*, 16(4), July 1986.

[Ber87] Robert Bernecky. An introduction to function rank. *ACM SIGAPL Quote Quad*, 18(2):39–43, December 1987.

[Ber91a] Robert Bernecky. Fortran 90 arrays. *ACM SIGPLAN Notices*, 26(2), February 1991.

[Ber91b] Robert Bernecky. Fortran 90 arrays. *APL-CAM Journal*, 13(4), October 1991. Originally appeared in *ACM SIGPLAN Notices*, 26(2), February 1991.

[Ber93a] Robert Bernecky. Array morphology. *ACM SIGAPL Quote Quad*, 24(1):6–16, August 1993.

[Ber93b] Robert Bernecky. The role of APL and J in high-performance computation. *ACM SIGAPL Quote Quad*, 24(1):17–32, August 1993.

[BH91] Robert Bernecky and R.K.W. Hui. Gerunds and representations. *ACM SIGAPL Quote Quad*, 21(4), July 1991.

[BI80] Robert Bernecky and Kenneth E. Iverson. Operators and enclosed arrays. In *APL Users Meeting 1980*. I.P. Sharp Associates Limited, 1980.

[Cam89] Lloyd W. Campbell, editor. *Fortran 88: A Proposed Revision of FORTRAN 77*. ISO/IEC JTC1/SC22/WG5-N357, March 1989.

[Can92] David C. Cann. Retire Fortran? A debate rekindled. *Communications of the ACM*, 35(8), August 1992.

[DD90] M.J. Daydé and I.S. Duff. Use of parallel level 3 BLAS in LU factorization on three vector multiprocessors – the Alliant FX/80, the CRAY-2, and the IBM 3090VF. *1990 International Conference on Supercomputing*, June 1990.

[DDHD90] J.J. Dongarra, Jeremy Decroz, Sven Hammarlung, and Ian Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1), March 1990.

[HIM91] Roger K.W. Hui, Kenneth E. Iverson, and Eugene E. McDonnell. Tacit programming. *ACM SIGAPL Quote Quad*, 21(4), August 1991.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman, 1990.

[HSF92] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: A method for scheduling parallel loops. *Communications of the ACM*, 35(8), August 1992.

[IBM94] IBM. *APL2 Programming: Language Reference*. IBM Corporation, second edition, February 1994. SH20-9227.

[Ive62] Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., 1962.

[Ive91] Kenneth E. Iverson. *Programming in J*, 1991.

[Ive96] Kenneth E. Iverson. *J Introduction and Dictionary*, J release 3 edition, 1996.

[Lei92] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.

[LHKK79] C.L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh. Basic linear algebra subroutines for FORTRAN usage. *ACM Transactions on Mathematical Software*, 5(3), September 1979.

[McI91] D.B. McIntyre. Language as an intellectual tool: From hieroglyphics to APL. *IBM Systems Journal*, 30(4), 1991.

[MI89] Eugene E. McDonnell and Kenneth E. Iverson. Phrasal forms. *ACM SIGAPL Quote Quad*, 19(4), August 1989.

[MM89] M. V. Morreale and M. Van Der Meulen. Primitive function performance of APL2 Version 1 Release 3 (with SPE PL34409) on the IBM 3090/S Vector Facility. Technical Report Technical Bulletin No. GG66-3130-00, IBM Washington Systems Center, IBM Corporation, May 1989.

[MSA$^+$85] J.R. McGraw, S.K. Skedzielewski, S.J. Allen, R.R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. SISAL: Streams and iterations in a single-assignment language, language reference manual. Technical Report M-146, Revision 1, Lawrence Livermore National Laboratory, March 1985.

[Per79] Alan J. Perlis. Programming with idioms in APL. *ACM SIGAPL Quote Quad*, 9(4–Part 1):232–235, June 1979.

[Smi81] Bob Smith. Nested arrays, operators, and functions. *ACM SIGAPL Quote Quad*, 12(1), September 1981.

[Ste93] Guy L. Steele Jr. High performance Fortran: Status report. *ACM SIGPLAN Notices*, 28(1), January 1993.

[Sto87] Harold S. Stone. *High-Performance Computer Architecture*. Addison-Wesley, 1987.

[Wal91] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of ASPLOS*, 1991.

[Wil91] R.G. Willhoft. Parallel expression in the APL2 language. *IBM Systems Journal*, 30(4), 1991.