Parallel expression in the APL2 language

by R. G. Willhoft

This paper reports on an investigation of parallel expression and execution in the current APL2 language. The study covers a historical, theoretical, and empirical viewpoint. The parallel nature of APL is traced from its foundations in the Iverson notation to current problems in executing APL on parallel hardware. The paper discusses features of the APL language and its current implementations that limit taking advantage of parallel expressions. A survey of related topics from the work on APL compilers is also included. Each APL2 language construct is examined for potential parallel expression. The operations are grouped based on the possible parallelism exhibited by each operation, and the possible implementation of each group is discussed. Three APL2 applications are explored to determine the actual parallelism expressed in "real" APL2 code. These applications are chosen from distinct areas: graphics, database systems, and user interactive systems. The actual data passed as arguments to every operation are dynamically examined, and the information is collected for analysis. The data are summarized and results of the study are discussed.

In the last several years, APL has received attention as a language that can be used to express parallel algorithms. The primary interest has been in the ability of the language to express algorithms on vector or array arguments directly, eliminating the need for a programmer to convert them into sequential loops. The question to be addressed in this paper is: Given a powerful array language, how much parallelism is expressed implicitly? This study attempts to better understand the extent of parallel expression that is contained in typical APL2 applications.

The parallel nature of APL2 is investigated in two ways that make this paper unique from similar studies in the past. First, there is an emphasis on completeness. All APL2 primitives are examined for possible parallel execution. Next, there is an emphasis on gathering empirical information. This study measures real code to achieve a better understanding of the extent of parallel expression in "real" APL2 code.

Once the parallel nature of current APL2 is understood, this paper also answers two other questions: From a language viewpoint what items could be changed to increase the parallel expression in the language; and what lessons can be learned regarding the development of parallel interpreters for the current APL2 language?

The parallel nature of APL

APL: A parallel language. APL is a language that can be considered parallel since its very inception. Ken Iverson, in his original definition of A Programming Language,[1] defines a language that is at its very roots a parallel language. The *Iverson notation* (the name used to describe the notation in Iverson's book) was not intended to be implemented. However, APL and APL2 were developed directly from the concepts that he outlined.

The 25 years of APL history have been scattered with work that has attempted to extract and exploit

the power of the Iverson notation. Recently much of that work has been focused on using APL (or APL-like notation) on parallel machines. The advantage of APL for parallel applications was recognized as early as 1970 by Abrams:

> In general, APL programs contain less detail than corresponding programs in languages like ALGOL 60, FORTRAN, or PL/I.... While this aspect of APL often makes programs shorter and less intricate than, say, ALGOL programs, it also requires that an evaluator of APL be more complex than one for ALGOL, especially if such expressions are to be evaluated efficiently. On the other hand, a machine doing APL has greater freedom since its behavior is specified less explicitly. In effect, APL programs can be considered as descriptions of their results rather than as recipes for obtaining them.[2]

The following sections explore the history of APL as it relates to execution on parallel machines.

Types of parallel expression. Parallel expression can be classified in a number of ways. The terms *course grain* and *fine grain* have been used to distinguish the size of the tasks that are executed in parallel. *SIMD* (simple instruction stream, multiple data stream) and *MIMD* (multiple instruction stream, multiple data stream) concentrate on the nature of the instructions that are issued to perform the calculations, and *vector processor*, *array processor*, and *multiprocessor* tend to emphasize the difference in the machine architectures that are used for parallel execution. All of these terms interrelate and are often used interchangeably.

For the purpose of this work, four terms will be introduced that focus on the nature of the expression from which the parallelism is extracted. They are *data parallelism*, *algorithm parallelism*, *data-flow parallelism*, and *task parallelism*.

The first three types of parallel expression are implicit—parallelism is "implied" by the operation(s) specified instead of being explicitly stated by the programmer. Task parallelism is the one explicit parallel expression.

Data parallelism in APL. Data parallelism refers to the application of a single conceptual operation to a number of data items at the same time. Each of the operations is completely independent from the rest. Hillis has coined the term *data parallel* to distinguish the difference in parallelism that comes from simultaneous operations across large sets of data, rather than from multiple threads of control.[3] The key concept of this definition is the fact that the expression of parallelism comes from the specification of operations across sets of data.

Although Hillis connects the idea of threads of control to his definition, our use of the term will not make this connection. There are times that the execution of a single conceptual operation to a set of data items will require, or at least allow, independent and distinct algorithms to be run on the separate data items. Although the execution in this case is MIMD, instead of the SIMD implied in Hillis's definition, the expression of parallelism is still of the data parallel form.

The concept of arrays of data is not unique to APL. What sets APL apart is that arrays in APL are viewed as a unified whole, rather than a collection of individual data items.[4] This view is what made Iverson's work so powerful. Iverson also defined operations on arrays including element-wise application of functions, scalar extension, selection, reduction, and permutation operations. The power of these concepts has been recognized in the work on new parallel languages[5] and in the work to include parallelism in existing languages, such as FORTRAN.[6]

Brenner[7] outlines some of the considerations and advantages of implementing APL on an array processor similar to the Connection Machine.[8] Brenner recognized the potential of execution of scalar functions, scan, and reduction on a parallel processor. Brenner also gives a thumbnail sketch of how some other APL operations might be executed in parallel. He outlines methods for **compress**, **expand**, **grade up**, **reshape**, **rotate**, **take**, **drop**, **index of**, **member**, and **inner product**. Although this is an impressive list, it is only a small part of the operations that can be done in parallel, as will be shown in this paper.

The parallel execution of APL has not only been shown theoretically, but also has been implemented in several machines. The Analogic APL Machine, introduced in 1980, used the APL language to drive a vector processor. As Delo points out, "One important achievement of the project is running software ... that had been written in a standard programming language to run on a conventional

computer."[9] Even today this is an achievement that has been matched by few other parallel computer projects.

While the APL machine was specially designed for APL execution, most parallel hardware is not designed with APL in mind. However, APL seems well positioned to take advantage of the new hardware. For example, the IBM 3090* Vector Facility is a high-performance pipeline processor designed to significantly improve vector performance.[10] APL2 was one of the first languages to use the Vector Facility for the processing of vector (array) data. The close match between the expressiveness of APL2 and the processing of the IBM Vector Facility has led Brown to conclude "... in some senses, the IBM Vector Facility is a machine designed for executing APL."[11]

Algorithm parallelism in APL. Algorithm parallelism refers to operations that can exploit the relationships of the data items to allow execution in parallel. This is in contrast to the assumption of independence among the items in data parallelism. In this form of parallelism, it is the algorithm that is parallel in nature. The data must be viewed as one item.

Examples are sorts, FFTs (Fast Fourier Transforms), matrix inversions, and similar operations. In each of these cases there are suboperations that can be executed in parallel, but these operations must be coordinated and supervised by an overall plan.

Although this type of parallel expression can clearly be replaced by algorithms written using the other parallel expression methods, the power of the expressiveness is lost. The advantage of capturing algorithm parallelism at the language level is that it allows for different architectures to execute the operation as is best suited for the machine.

Data-flow parallelism in APL. Data-flow parallelism results from the flow of results of one operation to arguments of the next operation. Since often there are multiple arguments to a given operation, each of those arguments can be calculated in parallel. To exploit data-flow parallelism it is necessary to calculate the data dependence (both argument and result) of each calculation. Then the order of calculation can be generated and is usually represented graphically. This directed graph shows the operations that can be executed in parallel.

This type of parallelism is by far the most difficult for the programmer to detect and exploit using explicit parallel expression. And although it is difficult for the system to detect this parallelism, the benefits of doing so are well worth the investment.

Most of the work that has been done in the area of data flow in APL has been in three areas. The first is work that is being done on developing an APL compiler.[12,13] Clearly, data flow is necessary to understand the manipulation of data in APL so that it can be compiled. The second area of work is in the area of functional languages. Backus[14] understood the potential that APL had as a functional language. Many have attempted to exploit this potential, usually with the goal of being able to create a parallel language based on functional constructs.[15,16] Finally, there are some who have looked at data flow solely as a method of execution within the APL language.[2,17,18]

In this section some of the methods and results of the work in all three areas are presented. The goal is to present the relationships between the work and some common ideas.

Abrams[2] and Wakshull[17] both explored the area of lazy evaluation. In this form of evaluation, values for arguments are not calculated until they are needed by the function that references them. Abrams used this idea to eliminate calculation on data that were later to be discarded, a concept he called "drag-along." Wakshull, while not discussing the benefits, gives a method by which an entire line of code can be executed using only data-flow principles.

Both Wakshull[17] and Ching[13] discuss the concept that both the left and right arguments to a function can be calculated at the same time. They formalize this concept by showing how a single dyadic function call can be placed within a pair of PARBEGIN and PAREND statements.

Budd[12] shows the power of constructing a complete data-flow graph. By doing so he is able to make statements about the rank, shape, and type of data variables. Although this benefit is connected with the problems of compiling APL, the technique is useful for discovering a number of properties of APL code without actually executing the code. For example, this type of analysis would be useful in determining interference between the assignments of two functions.

Task parallelism. Task parallelism expresses parallelism as separate tasks that are started and stopped by the application. These tasks run concurrently and may or may not communicate and synchronize with each other. All other forms of parallel expression can be broken down into task parallelism. The implicit parallel expressions already discussed are methods of hiding these operations from the user of the language, and therefore freeing the user to concentrate on the expression, not the control, of parallelism.

Task parallelism concentrates on the starting, stopping, synchronization, and communication between processes (tasks) at a level at which the user retains control over these operations. Task parallelism is exhibited in APL2 in the area of shared variables.

Shared variables, and the concept of auxiliary processors, are the oldest parallel facilities in APL. The auxiliary processor in APL can be a process running in parallel with the current workspace evaluation. The processing in the auxiliary processor is asynchronous to the workspace processing. The synchronization of the workspace with a given auxiliary processor is done with the shared variable. The shared variable is also used to pass commands to the auxiliary processor and to receive results from that unit.

APL2 has expanded the power and use of shared variables in several ways. Most importantly APL2 now allows variables to be shared between individual APL2 workspaces. In addition, several new shared variable system functions have been introduced that allow for more flexible methods of polling and using the shared variables. It has been noted by Gerth[19] that shared variables allow parallel structures without adopting artificial constructs in the language.

Hindrances to parallelism. There are some hindrances to parallelism in APL. These items must either be eliminated from the language or their effects must be minimized.

Assignments and side effects. One of the major problems in trying to execute code in parallel is that side effects may be produced. A side effect is any change in the state of the machine during the execution of a function that can be observed outside the function. Typical examples are assignments, I/O, and implicit results (such as the change to $\square RL$ made during the **roll** and **deal** functions). Side effects hinder parallelism because the total behavior of the program must create the same side effects in the same order to be a proper parallel implementation. Tu and Perlis[16] eliminate assignment in their functional language based on APL.

Dynamic binding. Dynamic binding causes the names in APL programs to be bound to values based on the environment in which the function is called. Dynamic binding makes it difficult to determine, before actual execution, many of the particulars of a program's activities. This complicates the areas of determining parallelism and avoiding interference. The alternative to dynamic binding is static binding. Static or lexical binding causes the values to be bound to the names based on the environment in which the object is defined. This solves many of the problems of program analysis and is therefore required by much of the data-flow work.[12,15,16]

Branching. The danger of GOTOs (branches in APL) have long been known by programmers. Specifically, in the area of parallel execution, branching makes it difficult to determine the exact execution of a program. At least two methods have been presented to deal with this problem. Some simply do not allow branching.[16] Others allow branching but only evaluate parallelism inside basic blocks (the areas between branches).[13]

Lack of declarations. Finally, the lack of declarations in APL deprives the interpreter (or compiler) of knowledge that is often known to the programmer. Some have suggested including (optional) declarations.[12]

APL2 as a parallel language

APL2 is an inherently parallel language because almost all primitive operations are defined on arrays of objects. The following sections classify and discuss these primitive operations. Akl defines parallelism as follows:

> Given a problem to be solved, it is broken into a number of sub-problems. All of these sub-problems are now solved simultaneously, each on a different processor. The results are then combined to produce an answer to the original problem.[20]

The key to exploiting parallelism is finding independent subproblems to be solved. The following discussion of each of the classes establishes how

Figure 1 Monadic scalar functions

Ceiling	**Floor**	**Pi-times**
Conjugate	**Magnitude**	**Reciprocal**
Direction	**Natural logarithm**	**Roll***
Exponential	**Negative**	
Factorial	**Not**	

* See Reference 22.

Figure 2 Dyadic scalar functions

Add	**Greater than or equal**	**Nand**
And	**Less than**	**Nor**
Binomial	**Less than or equal**	**Not equal**
Circular functions	**Logarithm**	**Or**
Divide	**Maximum**	**Power**
Equal	**Minimum**	**Residue**
Greater than	**Multiply**	**Subtract**

Figure 3 Right scalar function

Index of

independent subproblems can be defined. This then gives the key to implementation of these operations on a broad spectrum of parallel machines. For example, these operations could be done one per processor on a SIMD machine, or assigned in groups (based on data location) on a MIMD machine.

Scalar functions. Scalar functions can be most easily defined as the ability of a function to operate on individual elements of an array in exactly the same way that they are applied to the entire array. In other words, the calculation of every individual data element is independent of the other.

The following paragraphs define in turn monadic and dyadic scalar functions. The discussion of dyadic scalar functions includes the concepts of scalar extension, and also introduces two new terms, right scalar function and left scalar function. The functions that fit each of these categories are listed. Finally, there are functions that are closely related to scalar functions but do not fit the strict definition. These are also presented.

Monadic scalar functions. The formal definition of a monadic scalar function[21] is any function that meets the following requirement:

```
(F R)[I] ↔ F R[I]
```

The heart of this definition is the fact that the calculation of each element is independent of any other and that the definition of the operation on the whole array is defined in terms of the operation of the function on the individual elements. The functions in Figure 1 are defined in APL2 as being monadic scalar functions.

Dyadic scalar functions. The definition of a dyadic scalar function[23] is very similar to the definition of the monadic case. A dyadic scalar function is any function that meets the following requirement:

```
(L F R)[I] ↔ L[I] F R[I]
```

Again the independence of the individual calculations can be seen. Figure 2 illustrates the dyadic scalar functions.

Scalar extension. Scalar extension in APL2 is defined as "If one argument is a scalar or a one-item vector, pair the scalar or one-item vector with each item."[24] This allows APL2 to express the concept implicitly that most parallel languages define explicitly as a "data broadcast." The advantage in APL2 is that the programmer does not need to express the broadcast as a separate operation.

Right scalar functions—Consider now the case that the left-hand argument is not a single item, so that scalar extension would take place, but rather a data structure that is needed by each application of the function to items in the right argument. Therefore what we desire is not a scalar broadcast, but rather an array broadcast. This concept is captured in the following definition. A function will be called a right scalar function if the following is true:

```
(L F R)[I] ↔ L F R[I]
```

Although the term and definition is new, the concept is already used in APL2 in the function shown in Figure 3.

Left scalar functions—In a similar way, any function that meets the following requirement will be called a left scalar function:

```
(L F R)[I] ↔ L[I] F R
```

Again, this concept is also already used in APL2 in the function shown in Figure 4.

Each. The operator **each** accepts a single function as an operand, and the resulting derived function is monadic or dyadic based on the valence of that function. **Each** changes the operation of the function such that the function, instead of being applied to the entire argument(s), is rather applied to each item of the argument(s). The combination of all of these applications is the result of the derived function. **Each**, when applied to any function, produces a derived function that is by definition a scalar function (see Figure 5).

However, to be applied in parallel, one additional criterion must be satisfied; each application of the function must be independent of the others. The practical implication of this is that the function that is used must be free of side effects. This is true of all primitive functions in APL2 except for **roll** and **deal**. But this is not true of user-defined functions in APL2 in general.

Scalar related functions. There are a number of functions in APL2 that, although not strictly scalar functions, still exhibit many of the characteristics of scalar functions. These are listed in Figure 6, and the following paragraphs provide a brief description of how they are related to scalar functions.

Find—**Find** can be defined in terms of the left scalar function **member**. Each item of the left argument is searched for in the right argument using the member. After each search the partial result is shifted to another processor, based on the shape of the left argument, and the next search done. Clearly this is a highly parallel operation.

Format—In all three **format** functions—default, **format by example**, and **format by specification**—there is a right scalar operation. In **format by example** and **format by specification** the formatting of each item in the right argument can be carried on completely independently of the other. Only when all of the items are formatted must the result be compiled to form a new matrix. However, even this

Figure 4 Left scalar function

Member

Figure 5 Scalar derived functions

Each (monadic)	**Each** (dyadic)

Figure 6 Approximately scalar functions

Bracket indexing	**Index**
Find	**Index with axis**
Format (default)	**Interval**
Format by example	**Without**
Format by specification	

operation is an operation that is performed along axes and can be done in parallel.

In the case of default formatting, there is an additional step of determining the format parameters for each column. This also can be done in parallel with each processor determining on its own the required size for its item. These can then be combined together in a process very similar to reduction along the first dimension.

Indexing—Indexing, both in its functional form and as bracket indexing, would be difficult, but rewarding, to implement in a parallel form. **Index** would be considered a result scalar function, that is, each item in the result can be determined using an independent calculation based on the arguments. First, several sequential steps would be completed. The shape of the result would have to be determined and the locations allocated. Next, each location in parallel could obtain the correct indices; then, based on the shape of the array being indexed, it could determine positions and finally get the value from that position.

Interval—**Interval** is also a result scalar function. **Interval** would be very easy to implement on any machine in which each processor could determine a unique ID (identification), and all the IDs are sequential. **Interval** could then be simply imple-

Figure 7 Reduction functions

Reduce	**Reduce N-wise**
Reduce with axis	**Reduce N-wise with axis**

Figure 8 Scan functions

Scan	**Scan with axis**

Figure 9 Product functions

Decode	**Inner product**
Encode	**Outer product**

mented as laying out the shape of the result and telling each processor to generate a number based on its ID.

Without—**Without** is defined[25] as follows:

$$L{\sim}R \leftrightarrow (\sim L{\in}R)/L$$

The **member** and **not** part of the definition (most likely combined as a single operation) can be considered to be a left scalar function as defined above. The parallel nature of **replicate** will be dealt with later.

Reduction and scan. **Scan** and **reduce** operations (Figures 7 and 8), like scalar functions, have been at the heart of APL since its inception. Their importance to parallel processing has also been clearly established. Steele has called them primitive parallel operations.[5] **Reduce** can be considered a subset of the **scan** operation where only the final value is considered to be important.

When defined on vectors, these operations are parallel only when the function that is being applied is associative,[26] so only the associative case will be dealt with here. Brenner,[7] along with many others, has outlined a method of doing the **scan** (and therefore **reduce**) in $2{\otimes}\rho X$ passes. This means, for example, that a million element vector can be scanned in 20 operations on a sufficiently parallel machine.

The placement of these operations in this classification is very difficult. They are placed here so that we can deal with their primary definition, that is, on vectors. However, they are often used on higher order arrays (for example, on matrices). When applied to matrices these operations exhibit two levels of parallelism. First is the parallelism outlined above. Second is the parallelism that is involved in the application of the operation along an axis, as outlined in the next section. These two levels of parallelism can be handled separately, or combined to generate a highly parallel construct.

Product functions. The final set of functions that must be considered before we leave the area of scalar functions is the product functions (Figure 9). These functions are based on the dot operator.

Decode and **encode** are included with the product functions because they can be expressed in terms of the product functions.

The product functions are also result scalar functions, with each item in the result being calculated from a separate calculation. In the case of **outer product** this is the simple application of a function to two data items. In the case of **inner product** this result is more complex, consisting of the application of a function on two vectors and applying **reduce** to the resulting vector.

Axis functions. Moving from the area of scalar functions, the next logical step would be functions that are applied to subarrays of the arguments. These will be called axis functions. However, before presenting the individual functions, it would be helpful to formalize the concept of an operation along an axis and the concept of subarrays.

Subarray. A subarray is a subset of the data contained in an array that is selected by using zero or more elided axes. All nonelided axes must have a scalar value.

In APL2 the axis specification can be used to apply the function to independent subarrays within an array. The axis specified indicates the axis that is to be elided. We shall demonstrate this principle by discussing **enclose with axis** and **disclose with axis**. These functions were chosen because they can be used to describe all other operations that take an axis specification (see Table 1).

Table 1 Decomposition of axis specifications. The columns in this table show a decomposition of each of the lines of code that can be used to replace the most general case for each of the functions with axis specification.

Function	Result	Disclose Operation	Enclose Operation for Left Argument	Operation	Enclose Operation for Right Argument
Catenate	Z←	⊃[A]	(⊂[A]L)	,⍤⍤	⊂[A]R
Expand	Z←	⊃[A]		L\⍤⍤	⊂[A]R
Partition	Z←	⊃[A]	(⊂L)	⊂⍤⍤	⊂[A]R
Reduce	Z←	⊃		O/⍤⍤	⊂[A]R
N-wise reduce	Z←	⊃[A]	L	O/⍤⍤	⊂[A]R
Replicate	Z←	⊃[A]		O/⍤⍤	⊂[A]R
Reverse	Z←	⊃[A]		⌽⍤⍤	⊂[A]R
Rotate	Z←	⊃[A]	L	⌽⍤⍤	⊂[A]R
Scan	Z←	⊃[A]		O\⍤⍤	⊂[A]R
Drop	Z←	⊃[A]	(⊂L)	↓⍤⍤	⊂[A]R
Index	Z←	⊃[A]*	(⊂L)	⌷⍤⍤	⊂[A]R
Laminate	Z←	⊃[⌈A]	(⊂⍤⍤L)	,⍤⍤	⊂⍤⍤R
Ravel	Z←	⊃[↑A]		,⍤⍤	⊂[A]R
Scalar	Z←	⊃[A]	(⊂[A]L)	O⍤⍤	⊂R
Scalar	Z←	⊃[A]	(⊂L)	O⍤⍤	⊂[A]R
Take	Z←	⊃[A]	(⊂L)	↑⍤⍤	⊂[A]R

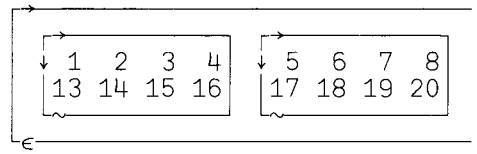
* See Reference 32.

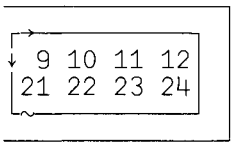
The **enclose with axis** function takes subarrays along the axes specified and makes them a single data item in the result. Therefore, the resulting matrix has the shape of the argument with the specified axes removed, and each item has the shape of the axes removed. For example:

```
      A←2 3 4⍴⍳24
      B←⊂[1 3]A
      ⍴B
3

      ⍴1⊃B
2 4

      DISPLAY B
```





The **disclose with axis** function is very similar to this except the elements are disclosed and placed back in the subarrays as specified. For example:

```
      C←⊃[1 2]B
      ⍴C
2 4 3
      C
 1 5 9
 2 6 10
 3 7 11
 4 8 12

13 17 21
14 18 22
15 19 23
16 20 24
```

For each of the axis functions listed in Figure 10, the application of the function results (conceptually) in the enclosing of the array along the given axis, applying the function to each item of the result, and then disclosing the result along the same axes. In light of parallel operation, it can be considered that each of the operations on the subarrays is an independent operation, and therefore can be done in parallel.

Scalar functions with axis. In addition to the above operations that take an axis specification, all of the scalar functions can take an axis specification. The concept is also based on subarrays and can be expressed in terms of **enclose** and **disclose** (see Table 1). The axis specification on scalar functions causes the items in one argument to be broadcast (scalar extension) to subarrays in the other argument.

Figure 10 Axis functions

Catenate	**Ravel with axis**
Catenate with axis	**Replicate**
Disclose	**Replicate with axis**
Disclose with axis	**Reverse**
Drop with axis	**Reverse along first axis**
Enclose with axis	**Reverse with axis**
Expand	**Rotate**
Expand with axis	**Rotate with axis**
Laminate	**Rotate along first axis**
Partition	**Take with axis**
Partition with axis	

Figure 11 Recursive functions

Depth	**Enlist**	**Match**

Figure 12 Matrix inversion functions

Matrix divide	**Matrix inverse**

Axis operators. Bernecky[27] and Gfeller[28] have both described a language enhancement called axis operator. Although their descriptions are different in syntax, they both carry the same fundamental idea. The axis operator has the effect of dividing the arguments into smaller matrices and applying the function to these smaller items. This type of operator would allow functions to be considered as axis functions, independent of their original type, much as the **each** operator forces its operand to be considered as a scalar function.

Recursive functions. Some functions in APL2 can be defined in the form of the following recursive definition:

f(x) = g(f applied to each item in x)

Where: f(x) is a function that is defined at some level in the tree (usually simple scalars)
g(x) is a combining function

The characteristic nature of these functions is that their execution results in a tree structure. The calculation in each of the branches of the tree is independent of the others and therefore can be done in parallel. The functions are shown in Figure 11.

Depth. The function **depth**, which returns the depth of the deepest item in a nested array, can be expressed in terms of a recursive definition:

$\equiv R \leftrightarrow 1+\lceil/\equiv{}^{\bullet\bullet}R$

Where: $\equiv$ *simple scalar* $\leftrightarrow 0$

Enlist. **Enlist** converts a nested array into a simple vector using a **depth** first method. The recursive definition of this routine is:

$\in R \leftrightarrow \supset,/\in{}^{\bullet\bullet}R$

Where: $\in$ *simple scalar* $\leftrightarrow$ *one item vector*

Match. **Match** returns a 1 if the two structures are identical at all levels, and a 0 otherwise. The recursive definition of this routine is:

$L\equiv R \leftrightarrow \wedge/L\equiv{}^{\bullet\bullet}R$

Where: $L\equiv R \leftrightarrow 0$ if L and R have different shapes
$L\equiv R \leftrightarrow 0$ if L and R are *simple scalars* with different values
$L-R \leftrightarrow 1$ if L and R are *simple scalars* with the same value

In general, the execution speed of **match** can be improved if, when any nonmatching condition is detected, all the execution in the tree is terminated and the 0 result returned. This makes the execution of the branches nonindependent, but they still can be executed in parallel.

Whole array functions. Moving from scalar to subarrays, the next logical step would be operations that manipulate entire arrays and therefore do not contain simple independent operations. However, both of these operations (see Figure 12) have been studied as classic parallel programming problems with many already published solutions.

The sorting functions in APL2 (see Figure 13) take an array as an argument and return a vector of indices as a result.

Rearrangement functions. The last class of operations that can be executed in parallel are those that

deal with data rearrangement. The characteristic of each of these functions is that the operation is done on addresses and not on data. The operations are shown in Figure 14.

The method of execution for each of these operations is basically the same:

1. Calculate the shape of the result.
2. Create an array of processors that match this shape.
3. Broadcast the control information to each processor.
4. Each processor calculates the current position of the data that are needed at that processor.
5. Each processor gets the data.

Not parallel. Some operations in APL2 cannot be executed in parallel. The primary reason for this is that they are defined on single objects or they do only a single operation. These operations are shown in Figure 15.

For example, **deal** is only defined on scalars. **Enclose**, **first**, **pick**, and **shape** all do a single operation on an entire array. **Execute** executes only one vector at a time. However, that line could be a parallel operation.

Other possible parallelism. There are other areas of possible parallelism in APL2. These are not discussed in this paper but are mentioned here for completeness.

Vector notation. Vector notation, or strand notation, allows a vector to be created by placement of objects next to each other. When these objects are simple constants, then creation of the vector is very straightforward. However, if the objects are expressions involving calculations, then this very simple construct allows for expression of a fork and join parallel structure.

Data-flow analysis. Data dependence is key to detecting parallelism in programs.[29] Several authors[12] have explored some of the areas of data-flow evaluation in APL. Most of this work has been related to the work being done on APL compilers.

Measurement of parallelism in APL2 code

For the measurements on the degree of parallelism, three applications were selected. These were selected to cover a broad spectrum of applications from the commercial data processing field. Each of the applications studied represents real code either available as a product or running in a manufacturing support area. Each of the applications is described briefly below, along with an explanation of the distinctions of that application.

Figure 13 Sorting functions

Grade down	
Grade down (w/collating sequence)	
Grade up	
Grade up (w/collating sequence)	

Figure 14 Rearrangement functions

Drop	**Transpose** (general)
Reshape	**Transpose** (reversed axes)
Take	

Figure 15 Functions that cannot be executed in parallel

Deal	**First**	**Ravel**
Enclose	**Pick**	**Shape**
Execute		

Database application. The first application studied was a database verification process. In this process approximately 5000 database records are read and all of the data in those records are verified. The information is verified by checking for consistency against lookup tables and checking for conformance to established input formats. The database is also conditioned to conform to the requirements for later processing.

This application was selected to show APL2 working in a non-numeric processing intensive process. The processing involves a large number of searches, sorts, justifications, and merges.

Interactive application. An education catalog and enrollment system was selected as an example of an interactive application. This system was highly user interactive, being completely full screen and menu driven. Within the application all user input is

checked for errors. During the session studied, the users searched the catalog using two different methods, viewed two course descriptions, enrolled in a course, scheduled time in a learning center, and reviewed their current enrollments.

This program contains a large amount of control flow logic code, which decodes user commands and performs complex error checking. Also, since it is a full-screen design, it must create and refresh screens and windows. The application also does a significant amount of formatting of data to display in "nice" formats to the user. This application would be considered by most to be a highly sequential system.

Graphics application. The last of the three applications that was studied is the GRAPHPAK workspace that is distributed with APL2. This workspace does a variety of presentation, business, and scientific/engineering graphics. The DEMO program within this workspace was used for the measurements on this application. This code represents fairly old APL code (late 1970s) that was written long before any emphasis on parallel processing.

The GRAPHPAK workspace uses APL functions to manipulate vector represented images and display them using GDDM (graphical data display manager). It uses homogeneous coordinates to perform a number of scaling and rotation calculations on graphical images. It is a concentrated use of the numeric capabilities of APL2.

Description of method. To measure the data parallelism in APL2 it was necessary to collect statistics regarding the data passed as arguments and operands during actual APL2 operation. The method chosen for this was to replace every primitive function and operator call with a call to a function or operator that would produce the same results but would collect information regarding the data passed to the operation. This method is outlined below.[30]

Workspace conversion. The workspace conversion consisted of replacing each primitive function and operator call, and all uses of brackets with calls to user defined functions. Each of these replacement functions had to fulfill two distinct purposes. First, it had to do exactly the same data manipulation as the primitive function. Second, it had to collect data and save the data for future use (see the next section on data collection). The two actions must be totally isolated from each other.

The first part of the replacement operation is easy in most cases. Most of the time it is possible simply to call the function that is being replaced. However, there are some cases that present problems. The replacement functions must explicitly handle fill and identity functions for empty arguments. Also bracket indexing and bracket axis must be implemented using the syntax of normal functions and operators. Finally, the **outer product** must be implemented as a monadic operator.

The workspace conversion consisted of replacing each primitive function and operator call.

A set of conversion routines was created that replaced all the primitive operations, as listed above, to the replacement routines. Often this was a simple replacement, but sometimes it involved syntactic changes to the code. For example, all bracket indexing were converted to the **index** function.

The converted workspace was shown to be the functional equivalent of the original workspace through a variety of verification methods. This converted workspace could then be run and the data collected automatically during operation.

Data collection. Each replacement function also must collect data. Each function evaluates its arguments, summarizes the information based on the operation type, and passes the information to the $\Delta\,\Delta COLLECT$ function. The $\Delta\,\Delta COLLECT$ function is responsible for compiling the information using several global variables. It is important that the data collection function interfere as little as possible with the application workspace.

The data were collected using a tabular method. A three-dimensional array was created with each plane being the information for one of the primitive

Table 2 Database application parallelism

Group Name	Total Calls to Operation	Primary Parallel Dimension			Secondary Parallel Dimension		
		Parallel Operations	Average Data Items	Maximum Data Items	Parallel Operations	Average Data Items	Maximum Data Items
ASCALAR	4,371	3,763	353	16,384	3,904	14	4,096
AXIS-A	10	10	2	2	10	8	8
AXIS-V	6,005	3,121	105	16,384	2,222	32	2,048
DERSCAL	12	0			0		
DSCALAR	2,781	343	181	8,192	0		
MSCALAR	206	63	60	128	0		
NOTPAR	1,751	1,470	3	1,024	20	31	256
PRODUCT	1,137	1,133	37	8,192	1,061	1,310	524,288
REARRANGE	3,786	1,096	7,064	524,288	3,303	3,048	524,288
RECURSE	10	10	9	16	0		
REDUCE	174	157	61	2,048	65	82	2,048
SCAN	70	70	44	64	1	8	8
SORT	0	0			0		

Table 3 Interactive application parallelism

Group Name	Total Calls to Operation	Primary Parallel Dimension			Secondary Parallel Dimension		
		Parallel Operations	Average Data Items	Maximum Data Items	Parallel Operations	Average Data Items	Maximum Data Items
ASCALAR	10,370	5,156	239	16,384	4,770	27	256
AXIS-A	218	218	2	16	71	6	64
AXIS-V	30,428	13,027	17	4,096	2,125	45	1,024
DERSCAL	1,879	1,650	4	32	0		
DSCALAR	18,022	2,906	21	1,024	140	2	2
MSCALAR	3,355	1,214	10	32	0		
NOTPAR	17,360	4,944	55	16,384	3,611	25	4,096
PRODUCT	1,627	1,515	8	64	276	247	2,048
REARRANGE	15,465	14,246	331	16,384	11,493	406	16,384
RECURSE	3,122	2,690	248	16,384	371	2	3
REDUCE	1,475	1,221	17	64	161	183	512
SCAN	307	282	22	64	3	9	16
SORT	14	14	8	16	13	12	32

operations. The arguments to the function are tabulated according to their primary and secondary parallel dimensions as in the table. The data are then tabulated in the array in groups; 0–8 have their own group and after 8 they are grouped by powers of 2.

The data collection routine also collects data on routines that either do not fit the above method or require more information to be saved. These are called exception data. All of these data are gathered during the operation of the application and then saved when the program is done.

Data analysis. The data are summarized by groups of operations. For each group, the following data are calculated:

- Total calls to operation—The total number of times that the operations in the group were called during running the application

For both the primary and secondary parallel dimensions:

- Parallel operations—The number of times the given operation(s) were called with two or more data items
- Average data items—The average number of data items for all parallel calls
- Maximum data items—The maximum number of data items presented to this operation by any single execution

Table 4 Graphics application parallelism

Group Name	Total Calls to Operation	Primary Parallel Dimension			Secondary Parallel Dimension		
		Parallel Operations	Average Data Items	Maximum Data Items	Parallel Operations	Average Data Items	Maximum Data Items
ASCALAR	27,393	13,957	16	2,048	10,541	63	512
AXIS-A	651	651	2	2	637	52	1,024
AXIS-V	27,587	20,905	22	2,048	4,914	19	1,024
DERSCAL	0	0			0		
DSCALAR	80,584	15,416	16	2,048	0		
MSCALAR	3,663	1,275	47	2,048	0		
NOTPAR	9,484	4,764	2	1,024	5	28	128
PRODUCT	3,026	2,912	14	1,024	1,387	39	2,048
REARRANGE	21,620	13,195	47	2,048	13,142	40	2,048
RECURSE	0	0			0		
REDUCE	6,235	5,899	10	512	1,276	16	1,024
SCAN	205	172	9	128	135	6	16
SORT	284	0			284	48	512

Table 5 Overall application parallelism

Group Name	Total Calls to Operation	Primary Parallel Dimension			Secondary Parallel Dimension		
		Parallel Operations	Average Data Items	Maximum Data Items	Parallel Operations	Average Data Items	Maximum Data Items
ASCALAR	42,134	22,876	122	16,384	19,215	44	4,096
AXIS-A	879	879	2	16	718	47	1,024
AXIS-V	64,020	37,053	27	16,384	9,261	28	2,048
DERSCAL	1,891	1,650	4	32	0		
DSCALAR	101,387	18,665	20	8,192	140	2	2
MSCALAR	7,224	2,552	30	2,048	0		
NOTPAR	28,595	11,178	26	16,384	3,636	25	4,096
PRODUCT	5,790	5,560	17	8,192	2,724	555	524,288
REARRANGE	40,871	28,537	458	524,288	27,938	546	524,288
RECURSE	3,132	2,700	247	16,384	371	2	3
REDUCE	7,884	7,277	12	2,048	1,502	37	2,048
SCAN	582	524	21	128	139	6	16
SORT	298	14	8	16	297	47	512

Since the information for some operator calls was included in the exception data, this information is also summarized.

Results. The results for each of the applications above are summarized in the following tables: Table 2, database application; Table 3, interactive application; and Table 4, graphics application. Table 5 shows the combined results.[31]

General observations. The percentage of parallel operations (approximately 45 percent of the 300K+ operations) is high. The average number of data items is moderate, 10–100.

It is interesting to note that the array operations force the user to write array code, hence there is a very high percentage of parallel operations. However, the scalar operations, especially dyadic scalar functions, which allow the user to write scalar code, have a much lower parallel operations count.

Application-specific observations. The database application exhibits the highest percentage of parallel operations (56 percent) and the highest average parallel operations (as high as 7K). The graphics application has lower average parallel operations than might be expected. This might be due to the fact that when this system was written, machines were smaller, and looping solutions were often used where array solutions would be used today. The interactive solution exhibited a higher than expected degree of parallelism.

Conclusions

In the introduction to this paper, the question of how much parallelism is expressed implicitly in APL2 was presented. This study shows clearly that APL2 exhibits a high degree of parallelism in its structure. APL2 is a parallel language due to a historical perspective that placed a high emphasis on array operations. The paper establishes that 94 of the 101 primitive APL2 operations can be implemented in parallel. We demonstrate also that typically 40–50 percent of APL2 code in "real" applications is parallel code. In light of these statistics, it is clear that APL is already a powerful parallel language.

Language recommendations. There are some features of the language that reduce the available parallelism. The following recommendations address specific areas in the APL2 language that increase the potential for parallel operation.

Side-effect-free functions. It has been shown that **each** is a highly parallel construct that can be used as a fork-join construct. However, for defined functions this construct cannot be executed in parallel. This is the result of the lack of side-effect-free functions in APL2. It is necessary for the programmer in APL2 to be able to declare, or have the system detect, that a given function has no side effects. Once this is done, it will be possible to parallelize expressions involving **each** (and other operators) without worrying about data interference.

Axis specification on more operations. Only a subset of the APL2 operations currently accept an axis specification. Because of this it is necessary for the programmer to manipulate the data before and after the operation to make the data conform to the required axes. Often the programmer will use an explicit **enclose-disclose** pair, use **transpose**, or (worse still) write a looping solution. This problem could be solved in two different ways. APL2 could be modified so that all, or at least most, operations accept an axis specification. Or, as has been proposed by others, an axis operator could be introduced.

Control flow operators introduced. **Each** is the first of several necessary control flow operators, essentially implementing a FORALL construct. Other operations need to be introduced to perform other structured processing constructs. For example, looping, recursion, if-then-else, case, and the WHERE construct from FORTRAN 8X need to be included. In addition to the obvious data-flow simplification, the APL2 language with these constructs (given a proper implementation) would be a much easier language to read.

Parallelization of APL2. In addition to the language considerations, this study leads to some conclusions in the area of execution of current APL2 on parallel machines.

Emphasis on array functions. Much of the emphasis in parallelizing APL2 has been with the scalar functions. This study points out that the array functions also provide a rich resource for parallelization. By considering the rearrangement functions as parallel operations on the addresses of data, rather than the data themselves, a large pool of parallel operation is available.

Importance of data-flow analysis. Finally, it is important to note that although 45 percent of the calls in this study could be executed in parallel, there is still a large body of code that is sequential in nature. Data-flow analysis is the key to unlocking the parallelism in this code. Much emphasis must be placed on this area of research if APL2 is to prove a successful parallel programming language.

Cited references and notes

1. K. E. Iverson, *A Programming Language*, John Wiley & Sons, Inc., New York (1962).
2. P. S. Abrams, *An APL Machine*, Ph.D. dissertation, Stanford University, Stanford, CA (February 1970), p. 64.
3. W. D. Hillis and G. L. Steele, Jr., "Data Parallel Algorithms," *Communications of the ACM* **29**, No. 12, 1170–1183 (December 1986).
4. Abrams (see Reference 2) shows how to do some array manipulations without any movement or calculation involving the actual elements. This concept involved the use of what he called *array descriptors*.
5. G. L. Steele, Jr., "Design of Data Parallel Programming Languages," presented at Syracuse University, NY (December 7, 1988).
6. "American National Standard for Information Systems Programming Language FORTRAN," Draft S8, Version 112, June 1989, *FORTRAN Forum* **8**, No. 4 (December 1989).
7. N. Brenner, "APL on a Multiprocessor Architecture," *APL82 Conference Proceedings, APL Quote Quad* **13**, No. 1, 57–60, ACM, New York (September 1982).
8. W. D. Hillis, "The Connection Machine (Computer Architecture Based on Cellular Automata)," *Physica* **10D**, 213–228 (1984).
9. J. Delo, "A High-Performance Environment for APL,"

APL84 Conference Proceedings, APL Quote Quad **14**, No. 4, 122–129, ACM, New York (June 1984).
10. R. S. Clark and T. L. Wilson, "Vector System Performance of the IBM 3090," *IBM Systems Journal* **25**, No. 1, 63–82 (1986).
11. J. A. Brown, "An APL2 Description of the IBM 3090 Vector Facility," *APL88 Conference Proceedings, APL Quote Quad* **18**, No. 2, 44–48, ACM, New York (March 1988).
12. T. A. Budd, "Dataflow Analysis in APL," *APL85 Conference Proceedings, APL Quote Quad* **15**, No. 4, 22–28, ACM, New York (1985).
13. W.-M. Ching, "Automatic Parallelization of APL-Style Programs," *APL90 Conference Proceedings, APL Quote Quad* **20**, No. 4, 76–80, ACM, New York (July 1990).
14. J. Backus, "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM* **21**, No. 8, 613–641 (August 1978).
15. A. Koster, "Compiling APL for Parallel Execution on an FFP Machine," *APL85 Conference Proceedings, APL Quote Quad* **15**, No. 4, 29–37, ACM, New York (1985).
16. H.-C. Tu and A. J. Perlis, "FAC: A Functional APL Language," *IEEE Software* **2**, 37–45 (January 1986).
17. M. N. Wakshull, "The Use of APL in a Concurrent Data Flow Environment," *APL82 Conference Proceedings, APL Quote Quad* **13**, No. 1, 367–372, ACM, New York (September 1982).
18. J.-J. Girardot, "The APL90 Project: New Dimensions in APL Interpreters Technology," *APL85 Conference Proceedings, APL Quote Quad* **15**, No. 4, 12–18, ACM, New York (1985).
19. J. A. Gerth, "Toward Shared Variable Events—Implications of *□SVE* in APL2," *APL83 Conference Proceedings, APL Quote Quad* **13**, No. 3, 265–274, ACM, New York (March 1983).
20. S. G. Akl, *The Design and Analysis of Parallel Algorithms*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1989).
21. *APL2 Programming: Language Reference*, SH20-9227-3, IBM Corporation (1988), p. 54; available through IBM branch offices.
22. **Roll** is defined as a scalar function and for most practical applications can be considered a scalar function. However, it is important to note that it does not strictly fit the definition. Consider the following example:

```
      □RL←5000
      ?(⍳100)[5 10 15 20 25]
1 7 1 6 21
      □RL←5000
      (?⍳100);5 10 15 20 25]
5 5 5 5 21
```

The problem with **roll** is that it is a function with side effects, so the order of calculation is important. In most cases, the above can be ignored because in each case five random numbers are generated, independent of the order of calculation. Although this effect can usually be ignored on sequential machines, the introduction of parallel calculation of random numbers is a much more complex problem. (See Reference 33.)
23. See Reference 21, p. 55.
24. *Ibid.*, p. 58.
25. *Ibid.*, p. 250.
26. This is not strictly true. For example:

```
-/X ↔ +/X×(⍴X)⍴1 ¯1
```

So it can be seen that some functions (notably **subtract** and **divide**) can be rewritten into functions that are associative.
27. R. Bernecky, "An Introduction to Function Rank," *APL88 Conference Proceedings, APL Quote Quad* **18**, No. 2, 39–43, ACM, New York (March 1988).
28. M. Gfeller, "A Framework for Extensions to APL," *APL88 Conference Proceedings, APL Quote Quad* **18**, No. 2, 162–165, ACM, New York (March 1988).
29. M. Wolfe and U. Banerjee, "Data Dependence and Its Application to Parallel Processing," *International Journal of Parallel Programming* **16**, No. 2, 137–178 (1987).
30. R. G. Willhoft, "A Tool for the Empirical Study of the Execution of APL2 Primitive Operations," *APL Implementer's Workshop*, Syracuse University, NY (September 11–14, 1990). (Includes a complete description of the method and code used for the study.)
31. R. G. Willhoft, "Parallel Expression in the APL2 Language," *APL Implementer's Workshop*, Syracuse University, NY (September 11–14, 1990). (Includes more complete test results.)
32. This disclose must allow simple scalars as elements, i.e., if the argument to disclose is simple, then do nothing.
33. O. E. Percus and M. H. Kalos, "Random Number Generators for MIMD Parallel Processors," *Journal of Parallel and Distributed Computing* **6**, 477–497 (1989).

Accepted for publication June 21, 1991.

Robert G. Willhoft *IBM Information Systems Division, 1701 North Street, Endicott, New York 13760.* Mr. Willhoft is currently an advisory engineer in Systems Analysis Engineering, IBM Endicott. He is currently working on his Ph.D. from Syracuse University with an expected graduation of December 1991. His dissertation topic is *A Parallel Language for the Expression and Execution of Generalized Parallel Algorithms*. Mr. Willhoft received his M.S.E.E. degree from Syracuse University in 1984 and his B.S. from Geneva College in 1978.

Reprint Order No. G321-5449.