R. H. Lathwell

System Formulation and APL Shared Variables

Abstract: The problem of providing communication with APL programs was approached by formulating systems as collections of autonomous processors communicating on interfaces consisting of shared variables. This paper discusses the formulation of a theoretical APL system and cites experience with a prototype APL shared variable system which both uses and provides shared variable interfaces.

Introduction

This paper discusses the results of an investigation of the nature of communication between cooperating processes that was motivated by the desire to provide a communication facility for APL programs. The principal results are a formalization of shared variables as interfaces between processors, and a theoretical mechanism—a shared variable processor—based on a formalization of systems in general and of APL systems in particular. A companion paper [1a] contains a discussion of shared variables in the context of the design of APL.

APL\360 [1b] is an implementation of APL as an interactive system on the IBM System/360 family of computers. The system was originally designed with two primary uses in mind: as an aid in teaching [2], and for system design [3]. However, the language is suited for almost all aspects of data processing and, once an implementation was available, its use spread rapidly, despite the fact that it provided communication only with typewriter terminals. There was no facility for passing large volumes of information across the workspace boundary. The use of APL\360 was therefore restricted to applications where all necessary information could be entered from a terminal device and stored in fixed-size workspaces.

Most programming languages approach communication and storage problems by defining explicit communication primitives such as READ and WRITE to transfer information. These specialized primitives, used in conjunction with declarative statements and job control languages, result in programs which contain file-handling details irrelevant to the algorithm, and are strongly dependent on host operating systems and file structures. This approach was deemed inappropriate for APL because it conflicted with many of the principles that guide APL design [1a]; in particular, it conflicted with the requirement for machine-independent theoretical definitions of primitive functions.

The basis for communication with an APL program was established in 1964 [4] when autonomous processors, described in APL, were shown communicating through interfaces consisting of shared variables having no other special properties. The concept of communication between processes by means of shared variables has also been used as a theoretical basis for other communication schemes. For example, Dijkstra [5] used it to develop semaphores (which are specialized shared variables) and the *P* and *V* functions defined on them.

Systems, processors, and interfaces

A *system* can be formulated and constructed as a collection of interconnected processors, each of which is designed to do a specific job. For example, computers are usually constructed with arithmetic and data transfer functions incorporated in independent processors; channels transfer data between high-speed storage and input-output devices, concurrently with CPU processing.

A *processor* can be described by an APL program [4,6,7] which does not terminate, i.e., all branches are made to statements within the program. It operates on values supplied to it, and may also store a result on the *interface* with another processor.

The interface is represented in each processor as one or more variables which appear in both, i.e., *shared variables*. (Hardware systems are often constructed in this way. For example, in System/360, communication between central processors and channels is effected by

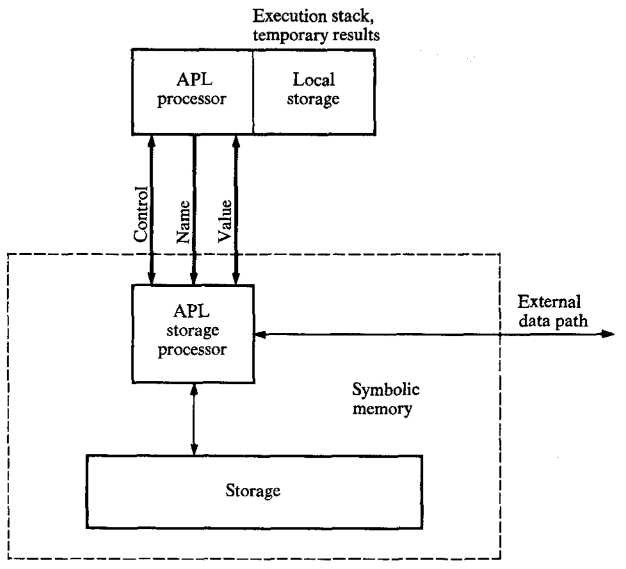

Figure 1 Organization of an idealized APL system.

variables such as the Channel Status Word and Channel Address Word, and by other interface variables located in local storage and set by the channel and by the execution of SIO, TIO, and HIO instructions in the central processor.) Either processor may read and write the shared variables, and in general, there is no other means of communication between them. When a processor is described by an APL program, the behavior of the interface is completely determined by the functions applied to the shared variables.

Some processors are designed to be servants to others. This design is achieved by constructing the processors so that their operation is directed by information supplied on an interface, with analysis and decision-making being performed by the master processor. For example, the IBM 1403N1 printer and IBM 2821 Control Unit are constructed so that the analysis is performed in the Control Unit. The printer receives interface values which specify print hammers to be selected, based on timing information supplied to the Control Unit by the printer. Systems are rarely constructed where one processor has access to the control mechanism of another, so that correct operation depends entirely upon precise specifications for the interfaces between the various processors.

The interfaces between the processors that form most conventional computing systems are usually permanently established when the system is installed, but as systems become larger and more complex it becomes increasingly desirable to have temporary interfaces that can be quickly connected and disconnected. For example, multiprogramming requires the dynamic allocation of subsets of a system; this is, in effect, the automatic construction of concurrent, temporary systems from a pool of available processors and other resources. Perhaps more important is the continuing development of telecommunications technology, making it both feasible and desirable to connect entire computing systems with one another, often on a transient basis.

An APL system

A theoretical APL system can be organized as shown in Fig. 1. The system consists of an APL *processor*, which executes all APL primitive and defined functions, and a *memory* which stores global and local variables and function definitions. The APL processor has its own local storage for information relevant to the execution state, including an execution stack and any temporary results that might be required during the evaluation of a statement. The memory, or *storage processor*, consists of a processor and a storage device. In the particular case of an APL memory, the contents are accessed by names that have no direct relationship to the locations at which information is stored—hence the memory is called a *symbolic memory*.

Information can be passed into or out of the APL system only by way of a variable called the *external data path*. One of the principal results of the present investigation is the design of a processor that is connected to the storage processor by sharing this variable with it, and acts as an intermediary in establishing and maintaining transient connections between the APL system and its environment. An organization for this *shared variable processor* is shown in Fig. 2. The organization is the same as that of the APL system, except for the location and number of external data paths.

Figure 3 illustrates how a multiple-user system like the (time sharing) APL\360 might be constructed using the shared variable processor. Each user has a separate APL system, connected to its environment by way of the external data path that is shared as shown in the figure. Part of this environment is a single APL supervisor that governs the connection of each APL system to its sources and repositories of information: the APL libraries, parts of the host system, and the users themselves at typewriter terminals.

The organization illustrated in Fig. 3 allows the APL system to be designed and constructed so that it is independent of the design of other aspects of the environment (such as the host operating system), and is dependent only upon the precise definition of the interface with the shared variable processor. It reduces the problem of providing a general communication facility for APL programs to the problem of providing the APL system with an explicit shared variable facility. This problem is approached in the following section by a more detailed examination of the APL storage processor.

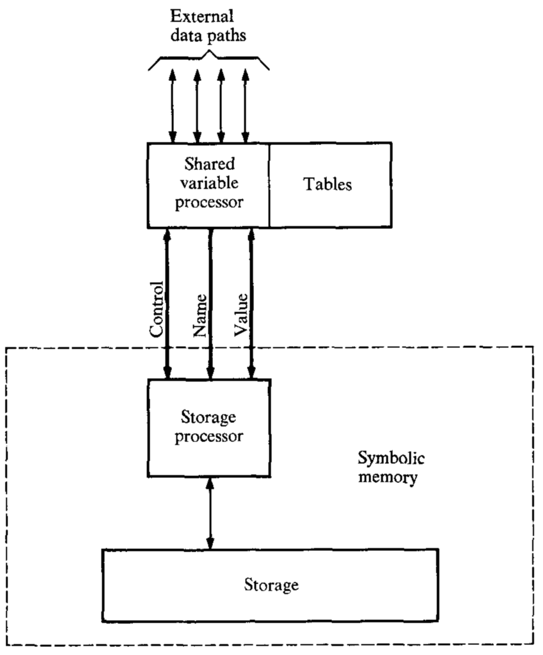

Figure 2 Organization of a shared variable subsystem.

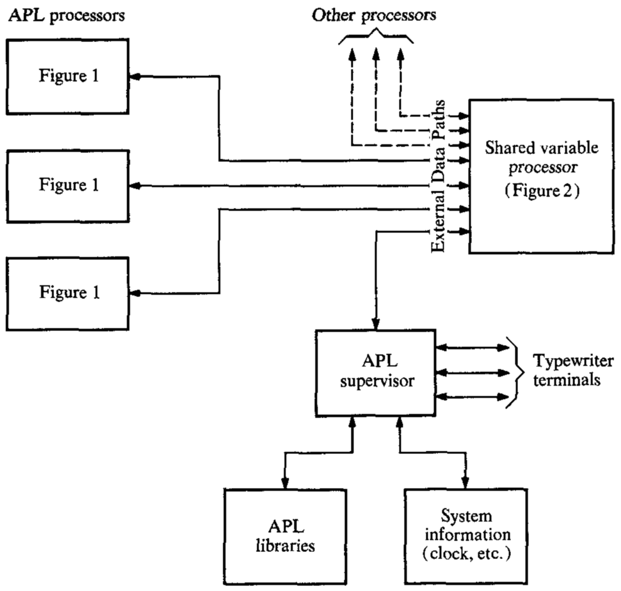

Figure 3 A theoretical construction of an APL time-sharing system using shared-variable interfaces.

APL storage

The APL storage processor determines where information is stored. It cooperates with the APL processor (which is, in theory, completely unconcerned with the storage process) and with the shared variable processor.

The program *STORAGE* in Fig. 4 is a simplified model of APL storage; a complete APL storage is somewhat more complicated since it stores functions and other information in addition to variables. It is also likely that a practical system would require additional information, such as temporary results and the execution stack, to be kept in the main storage rather than in the APL processor, as is assumed in the idealized system shown in Fig. 1.

In the function *STORAGE* (which is written in 0-origin) three shared variables, *CONTROL*, *NAME*, and *VALUE*, form the interface between the APL processor and the storage processor. It is presumed that the APL processor notifies the storage processor when access to a variable is required by setting an appropriate value (*R* or *W*) in the *CONTROL* variable, after placing necessary information in the *NAME* and *VALUE* variables. The storage processor attempts to honor the request and signals completion by setting a control value indicating either success or an error. The variables *NAMETABLE*, *CLASS*, and *ASSOCIATION* denote the tables used by the storage processor to keep track of stored information. *NAMETABLE* is a matrix containing the names used by the APL programs which are being executed by the APL processor, one per row. *CLASS* is a vector whose elements indicate the kind of object denoted by the names in the corresponding rows of *NAMETABLE*, and *ASSOCIATION* is a vector whose elements are the values associated with the names. (In practice, of course, *ASSOCIATION* would not be a vector of scalars but would be a more complex structure. It is represented as a vector here for simplicity in exposition.)

The kind of request is determined in statement [1],* and the processing of a WRITE request is shown in [3] through [18]. The name table is searched [3] and if the name is not found, a new entry is created and the value stored, [5] through [7]. If the name already exists, and is not a shared variable [9] but a variable [10], the value is replaced [11]. A signal indicating successful completion is then sent to the APL processor [12], and the storage processor returns to its idling state [1] until the APL processor requests another operation. A request to read is processed in statements [19] through [25]. The name table is searched [19], and if the name is not found [20], or is neither a shared variable [21] nor a variable [22], an appropriate error signal is returned to the APL processor [32], [33]. When a variable happens to be shared (detected at statements [9] and [21]), the storage processor passes the request to the shared variable processor.

The APL processor requires no knowledge of the definitions of names, and depends on the storage processor to take appropriate action and to signal with an error indication in *CONTROL* when a name does not denote a variable. The interactions between the storage con-

*Bracketed numbers in this section refer to the statements in Fig. 4.

```
      ∇STORAGE
[1]   WAIT:→(CONTROL=R,W)/READ,WRITE
[2]    →WAIT

[3]   WRITE:I←(NAMETABLE∧.=NAME)⍳1
[4]    →(I<1↑⍴NAMETABLE)/L
[5]    NAMETABLE←NAMETABLE,[0]NAME
[6]    CLASS←CLASS,VARIABLE
[7]    ASSOCIATION←ASSOCIATION,VALUE
[8]    →Z
[9]   L:→(SHVAR=CLASS[I])/SVW
[10]   →(VARIABLE≠CLASS[I])/SE
[11]   ASSOCIATION[I]←VALUE
[12]  Z:CONTROL←SUCCESS
[13]   →WAIT
[14]  SVW=EDP←W
[15]   EDP←NAME
[16]   EDP←VALUE
[17]   CONTROL←EDP
[18]   →WAIT

[19]  READ:I←(NAMETABLE∧.=NAME)⍳1
[20]   →(I=1↑⍴NAMETABLE)/VE
[21]   →(SHVAR=CLASS[I])/SVR
[22]   →(VARIABLE≠CLASS[I])/SE
[23]   VALUE←ASSOCIATION[I]
[24]   CONTROL←SUCCESS
[25]   →WAIT
[26]  SVR:EDP←R
[27]   EDP←NAME
[28]   →(SUCCESS≠I←EDP)/SVRZ
[29]   VALUE←EDP
[30]  SVRZ:CONTROL←I
[31]   →WAIT

[32]  SE:CONTROL←'SYNTAX ERROR'
[33]   →WAIT
[34]  VE:CONTROL←'VALUE ERROR'
[35]   →WAIT
      ∇.
```

Figure 4 A simplified model of APL storage.

troller and the shared variable processor consist of particular values of *EDP* that specify the kind of request, followed by a predetermined sequence of values which convey any additional information. An interface which depends on a sequence of values (or interplay) to effect a single request will operate correctly only if there is a mechanism to ensure that information will not be destroyed in the process. Such a mechanism is discussed later.

A request to write a shared variable is passed to the shared variable processor [14 through 17] by storing a WRITE signal in *EDP* [14], followed by the name [15] and value [16]. The storage controller then reads the result from the shared variable processor and passes it on to the APL processor [17]. The value of a shared variable is read by sending a READ signal [26] followed by the name [27]. The shared variable processor returns a result which is saved and checked [28], followed by the value if the operation was successful [29]. At successful completion of a READ, the storage processor passes the value [29] and the result indication [30] to the APL processor.

This description of the storage processor illustrates two ways to construct shared variable interfaces. The interface between the APL processor and storage consists of multiple variables, each with a specific meaning, while the interface between the storage and the shared variable processor consists of a single variable and a predetermined sequence for passing several related items of information. In practice, the techniques used will depend on the economics involved in a particular system, and a combination of approaches is often found. (See, for example, the interface between the System/360 CPU and channels in Ref. 4.)

This formulation of a storage processor, though incomplete in both function and detail, provides a sufficient understanding of the external data path interface for consideration of the design of a shared variable processor. A complete formal design depends only on the precise definition of the interface, i.e., the number of variables which form the interface (in this case, one), the permissible values, and the communication sequences.

The shared variable subsystem

The shared variable subsystem (Fig. 2) in principle consists of a specialized shared variable processor and a symbolic memory that is used to store shared variables. The shared variable processor has two control tables—a table with one entry for each processor connected, and a table with one entry for each shared variable stored. Each entry in the shared variable table contains the external name of the variable (i.e., the name supplied by the sharing processors), a unique internal name created by the shared variable processor, the identifications of the processors sharing the variable, and access control information.

The function of the shared variable subsystem is analogous to the function of a central switching office of a telephone system—it is activated when some processor wishes to be connected to another, it establishes and maintains the connection, and it terminates the connection when it is no longer needed. A processor *offers* to share a variable by submitting a request which includes the identification of the intended partner and the name of the variable. The shared variable processor searches its shared variable table, and if it finds a corresponding unmatched offer established by the partner at some earlier time, the connection is made by noting, in the table entry, that the offer has been matched (accepted). Otherwise, an internal name is created and the offer is entered in the shared variable table. A processor may disconnect or *retract* a shared variable at any time by withdrawing the offer to share. If the variable is connected when the request for retraction is received, an outstanding offer to the disconnecting processor is left, and a subsequent offer to reshare the same variable will result in a reconnection. If the variable is not connected, withdrawing the offer to share causes the entry to be deleted from the shared variable table.

In many applications, it is desirable to inhibit access to a shared variable to avoid repeated tests by one processor to determine whether its partner has accessed the variable. For example, the external data path, EDP, in the description of the APL storage processor shown earlier will not operate correctly unless successive WRITE attempts are inhibited until the partner has read each value, and successive READ requests are inhibited until the partner has supplied new values. The shared variable processor provides the necessary inhibition as a function of an *access control vector* that is associated with each shared variable.

In the following discussion, the two processors sharing a variable are identified as *A* and *B*. In the actual design of the shared variable processor, symmetry is imposed so that this distinction is invisible to the sharers. Information is presented to each so that it regards itself as system *A*.

The access control vector is a four-element vector of ZEROS and ONES, ONE meaning that a particular access is to be controlled, as follows:

ACV[0] – Two successive WRITES by *A* require an intervening READ or WRITE by *B*.
ACV[1] – Two successive READS by *B* require an intervening READ or WRITE by *A*.
ACV[2] – Two successive WRITES by *A* require an intervening WRITE by *B*.
ACV[3] – Two successive READS by *B* require an intervening WRITE by *A*.

The asymmetry (i.e., a READ inhibition is removed by a WRITE but a WRITE inhibition is removed by either a READ or a WRITE) ensures that a processor will not overwrite a value which its partner has not had the opportunity to read, but does allow the partner to ignore the value and to overwrite it. This facility allows a processor to deviate from a normal communication sequence when the value it wishes to send to its partner is independent of the next value to be passed by the partner.

The access control vector *ACV* is established by the expression *ACV*←*CVA*∨*CVB*[1 0 3 2], where *CVA* and *CVB* are the control vectors provided by *A* and *B*. The permutation of *CVB* imposes symmetry.

The shared variable processor also maintains an *access state vector*, *AS*, having the following significance:

0 0 – neither processor has written a value the other has not read.
0 1 – *A* has written and *B* has neither read nor written since.
1 0 – *B* has written and *A* has neither read nor written since.

The access control vector, the access state vector, and the type of access requested (READ or WRITE) together determine both the decision to permit or deny the requested access, and the new value of the access state. Using a ONE to denote a WRITE access and a ZERO to denote READ, the behavior may be described by the following function, which returns a ONE if access is denied:

```
     ∇ R←AC REQ
[1]    R←((2↑ACV)∧AS=0 1)[REQ]
[2]    →R/0
[3]    AS←0 1∧AS∧REQ∇
```

The shared variable processor has no control over the processors connected to it, and when it determines that a particular request is inhibited, it returns an appropriate signal on the external data path. The requesting processor will make whatever use of the information it wishes; it may, for example, wait for a signal indicating that the access state or access control vector has been altered by the partner, or it may retry later.

When a shared variable is established, the access control vector is all zero, the initial access state is 0 0, and the initial value is obtained from the first offerer.

Prototype implementation

In order to evaluate the use of shared variables in practical applications and to verify the theoretical design of the shared variable processor, a prototype was constructed and put into operation at the Philadelphia Scientific Center in early 1971. The prototype was designed as a separate processor very much like the theoretical processor described here, and the design was embodied in a collection of APL functions which ultimately provided a working model. Implementation of the prototype consisted in transcribing the APL model to System/360 assembly language and defining an interface analogous to the external data path.

The APL\360 interpreter was modified to introduce shared variables as a recognized class of objects and to incorporate system functions that provide APL users with the ability to offer and to withdraw shared variables and to specify access control.

The resulting system differed in some respects from the theoretical design. In particular, the APL\360 interpreter differs from the ideal organization shown in Fig. 1 in that the APL processor and the storage processor are combined. Instead of the interpreter being extensively redesigned so that shared variables could be incorporated as objects identifiable by the storage mechanism, it was augmented to incorporate the shared variables as a syntactic class. This, however, is not apparent to the user.

The prototype shared variable processor differs from the ideal in one principal respect: it was implemented as a program residing in the same physical computer as the APL system rather than as a separate machine. As a result, the amount of storage space available to it is constrained, and this constraint is seen in two ways—the maximum size of a single shared variable is restricted, and the value associated with a shared variable is deleted from the shared variable storage when both sharing processors are known to have read it.

Auxiliary processors

The original motivation for this project was the desire to provide APL users with a facility for accessing information contained in files. The formalization of shared variables and their introduction into the implementation provide sufficient language capability, but do not in themselves provide access to files.

Initial file applications required the ability to process files of existing data which had been produced by conventional systems. To provide access to these files, small programs in conventional languages were written to provide a shared variable interface to the APL user on one side and a standard Operating System access method interface on the other. The function of these programs was analogous to impedance matching of electrical interfaces, and they were the first of a class of programs, now called *auxiliary processors*, which have shared variable interfaces and are designed to perform services for APL programs, but because of some constraint cannot be written in APL.

It was at first thought that auxiliary processors should be highly specialized, so that very little related function would be required in APL programs. Experience so far indicates the opposite to be desirable, i.e., auxiliary processors should be simple, and the bulk of the function should be implemented in APL programs. A specific example concerns the use of high-speed printers. The first processor to support printers was a complex program that printed the value of a shared variable as it would appear when printed on an APL terminal. However, users soon showed a preference for the greater flexibility provided by standard sequential character output, which required all of the output to be formatted by them.

Experimental results

The prototype APL shared variable system has been in use for over two years and has been evaluated in a large trial number of situations. Applications have ranged from simple sequential input-output to the direct support of hardware devices. Conditions have varied from a dedicated system running a single application to a general data-processing system with more than one hundred simultaneous users.

The use of shared-variable interfaces as a basis for communication between cooperating processes has proved to be both a practical means of providing communication to APL programs and, since the technique was also used in the design of the system, to be a sound method of constructing systems.

The results of the evaluation of the prototype shared variable system can be summarized as follows:

1) The system is efficient. The processing time for a typical data processing application is competitive and often better than for the same application programmed in a conventional manner. The reasons for this are not entirely understood, but contributing factors appear to be low system overhead and easy programming. The latter permits the user to select the best of many trial solutions.
2) Limited storage does not cause problems, primarily because the average size of interface variables is small (measured over several million data transfers, the average variable required approximately 200 bytes) and because of the transient nature of interfaces.
3) The system is easy to use. Most of the applications so far implemented on the shared variable prototype have been programmed by professionals in fields other than programming, and many of these applications had not been previously implemented on any computing system. In cases where an application had been programmed on a conventional system, the effort required was at least an order of magnitude greater than in APL.
4) Shared variable interfaces impose a discipline that results in better system designs, since they force the designers to formulate the system as a collection of independent processors with well-defined interfaces. Each processor is then easy to implement, and the effort required to produce a total system is substantially reduced.

Unsolved problems

Shared variable interfaces and the shared variable processor have extended the applicability of APL, but they have also emphasized problems which remain to be solved:

1) The APL workspace has a finite fixed size. The availability of files for large volume storage alleviates this somewhat, but requires defined functions to be used in situations where an algorithm could otherwise be stated by a single APL primitive.
2) APL has no program-controlled interrupt facility. For example, execution errors always require manual intervention at the terminal. When a request for access to a shared variable is denied, APL either waits until access is permitted or reports an execution error. There is no way for an APL program to respond immediately to an asynchronous event.

Acknowledgments

The work reported here was done in close collaboration with L. A. Morrow. The prototype system, a valuable working tool, was produced by L. A. Morrow, J. A. Brown, and C. F. Shen. J. K. Tuttle and R. D. Shively developed the first general file processor, and the users of the APL system at the IBM Philadelphia Scientific Center provided an evaluation of the concepts.

References

1. (a) A. D. Falkoff, and K. E. Iverson, "The Design of APL," *IBM J. Res. Develop.*, **18**, 324 (1973, this issue). (b) A. D. Falkoff and K. E. Iverson, *APL/360 User's Manual*, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1968. IBM manual number GH20-0683.
2. K. E. Iverson, "The Use of APL in Teaching," Publication G320-0996, IBM Corporation, White Plains, New York 1969.
3. A. D. Falkoff, "Criteria for a System Design Language," *Report on NATO Science Committee Conference on Software Engineering Techniques*, 1970.
4. A. D. Falkoff, K. E. Iverson, and E. H. Sussenguth, "A Formal Description of System/360," *IBM Systems Journal*, **3**, 198 (1964).
5. E. W. Dijkstra, "Co-operating Sequential Processes," *Programming Languages* (F. Genuys, ed.), Academic Press, New York, 1968.
6. R. H. Lathwell and J. E. Mezei, "A Formal Description of APL," Colloque APL, 9–10, IRIA, Paris, September 1971.
7. H. Hellerman, *Digital Computer System Principles*, McGraw-Hill Publishing Co., Inc., New York, 1967.

Received June 28, 1972
Revised May 3, 1973

The author is located at the IBM Data Processing Division Scientific Center, 3401 Market Street, Philadelphia, Pennsylvania 19104.