NASA-CR-172,252

NASA Contractor Report 172252

ICASE

A MODEL FOR THE DISTRIBUTED STORAGE AND PROCESSING OF LARGE ARRAYS

Piyush Mehrotra

and

Terrence W. Pratt

Contract Nos. NAS1-17070, NAS1-17130
October 1983

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

A MODEL FOR THE DISTRIBUTED STORAGE AND PROCESSING OF LARGE ARRAYS

Piyush Mehrotra
Institute for Computer Applications in Science and Engineering

Terrence W. Pratt
University of Virginia

ABSTRACT

A conceptual model for parallel computations on large arrays is developed in this paper. The model provides a set of language concepts appropriate for processing arrays which are generally too large to fit in the primary memories of a multiprocessor system. The semantic model is used to represent arrays on a concurrent architecture in such a way that the performance realities inherent in the distributed storage and processing can be adequately represented. An implementation of the large array concept as an Ada package is also described.

This work supported in part by the National Science Foundation under grant MCS78-00763 while the authors were in residence at the University of Virginia and by NASA Contracts NAS1-17070 and NAS1-17130 while the authors were in residence at ICASE.

N84-12742#

1. Introduction

One of the major driving forces behind proposals for large-scale parallel multiprocessor architectures, such as CEDAR [4], MPP [3], TRAC [16], and Blue CHIP [17], has been the need for more processing power for large scientific and engineering applications. A major issue in the effective use of such systems is the design of the input/output aspects of the system, that is, the methods by which large quantities of data may be effectively moved from secondary storage into the system, routed to the appropriate processing elements for processing, and then back to secondary storage. For example, in the NASA Massively Parallel Processor (MPP), designed for processing LANDSAT-D satellite images, the processing array has over 16000 processing elements (128 x 128 array of PE's), but each image consists of a 6000 x 6000 array of pixels. To process such an image, more than 2000 blocks of data must be moved from secondary storage through the PE's and back to secondary storage.

In scientific and engineering problems, the large data objects of interest are usually arrays. Typically these arrays are too large to fit in the central memory of even a large sequential computer. In a multiprocessor, such an array may occasionally be distributed only across the primary memories of the PE's, but generally it must be partitioned between secondary storage and the primary memories.

Appropriate hardware and operating system software for managing the distributed storage and processing of large data objects in a parallel system are only part of the problem. Stevenson and Perrott [13] in a survey of the problems encountered in the use of ILLIAC-IV report that one of the major programming problems was the fact that the users had to resort to the use of relatively low-level assembly language programs to handle the "backing store traffic" required to move data in and out of the machine. If applications programmers are to make effective use of a parallel system, a high-level view of data storage and movement is needed at the applications level.

Unfortunately, there are major performance realities associated with data movement within a parallel system, both for data moving into and out of the system and for data moving between processing elements. A high-level abstraction should not hide these performance realities from the programmer, a point strongly emphasized by Jones and Schwarz [6] in their study of experience with multiprocessor programming. Thus what is required is a high-level abstraction that can reflect accurately the performance realities of a parallel system, so that the applications programmer can make effective use of the system without resorting to low-level primitives.

In this research, attention is restricted to arrays as the large data objects of interest, because arrays form the major large structures in many scientific and engineering applications. In this context there are two coupled problems for which we seek an effective language treatment:

(1) The partitioning of an array and its <u>distributed storage</u> on both secondary storage and processor memories in a multiprocessor architecture. To process a large data structure such as an array requires a complex series of data partitionings and data movements through the distributed system. We seek a solution that frees the user from the task of managing directly these partitionings and movements, without masking the performance realities involved.

(2) The partitioning of an array and its <u>concurrent processing</u> by the separate processors of the multiprocessor architecture. To effectively utilize a multiprocessor architecture, the total computational task must be divided into subtasks capable of executing in parallel. The subtasks need to access the data structures involved simultaneously so as to be able to run concurrently. The language should provide the appropriate higher level primitives to express the appropriate division and sharing of the data among these subtasks. These primitives should allow large-scale parallel computation on the data structure without a major overhead in subtask communication for the purpose of synchronizing access to the data structure. Subtasks should be able to traverse the data structure without unnecessary mutual exclusion from concurrent access.

The goal of this research is to develop a conceptual semantic model for parallel computations on large arrays that addresses these two issues and that can be effectively implemented on a variety of multiprocessor architectures. The next section provides some background for the study. The following section presents the model, with a rationale for the major design decisions. A general implementation strategy, presented as an Ada[1] package, together with an example of use of the model, follows. The characteristics of a particular architecture obviously give rise to different performance characteristics, and in some cases, necessary restrictions on the model. Implementation strategies for various typical multiprocessor architectures appear in [10]. Architectures based on shared memory and architectures with no shared memory between processors suggest somewhat different solutions to the design problem. The semantic model presented here allows an effective solution on both types of architecture.

2. Background

Languages for programming multiprocessor architectures have largely ignored the problems associated with concurrent processing of large data structures. For example, languages proposed for SIMD machines, e.g., PASCALPL [19], IVTRAN [11], and ACTUS [12], extend sequential languages such as PASCAL or FORTRAN to take advantage of the element-level parallelism exhibited by SIMD architectures. Thus a statement of the form:

```
A := B + C
```

where A, B, and C are arrays of the same size, implies that for all I

[1] Ada is a registered trademark of the Department of Defense

and J the following statement:

```
A[I,J] := B[I,J] + C[I,J]
```

is to be executed in parallel. For array sizes smaller than the size of the hardware matrix of processing elements, the above statement can often be executed in a single machine cycle. But if the sizes of B and C are larger than the hardware matrix, then they have to be partitioned into blocks before the computation can be carried out. Typically the languages provide only rudimentary constructs for making this partition and managing the movement of data between secondary storage and processing element memories.

For MIMD architectures, the programming problem is usually more difficult. The large data structure not only has to be distributed among processor memories and secondary storage, but the separate asynchronous tasks involved may require different size partitions of the data. Languages designed for MIMD architectures, such as CSP [5], ARGUS [8], and Ada [2], usually provide no special features for processing large data objects other than files. Data objects, whether large or small, are considered to be owned by a single task and the access to the data structure is controlled by the task. To effectively utilize a multiprocessor architecture while processing a large data structure in parallel, independent tasks need to have the capability of accessing independent parts of the data structure concurrently. Of course, two tasks have to be prevented from simultaneously updating the same portion of the data structure. Languages designed for MIMD processors do not provide the higher level view of distributed control of a single data structure while providing mutual exclusion between the tasks where appropriate.

Language concepts for file processing, in fact, are similar in important ways to those required for large array processing, as discussed in more detail in the next section. The concept of a "moveable window", in particular, is useful for both file and large array processing, where the window represents a part of the data object currently available for processing. The utility of "window" concepts for partitioning and processing arrays has occasionally been recognized in sequential languages, most notably in the language OL/2 [14]. In OL/2 a powerful and flexible mechanism for defining and using windows on arrays is developed, which allows an array to be partitioned for processing in very natural ways. The mechanism allows windows to be hierarchically decomposed into smaller windows to as many levels as desired. We develop a similar but simpler set of window concepts for the case of distributed storage and processing, a case not considered in the OL/2 design.

3. A Model for Array Processing

In defining a model for concurrent processing of large arrays, the roots of our approach are found in traditional language constructs for file processing rather than array processing. In sequential languages linear arrays and files are distinguished, although structurally each is a linear sequence of elements of some type. Files, however, have several characteristics that distinguish them from arrays. For this discussion, the most important are:

(1) A file is a distributed data structure, stored primarily on secondary storage, rather than in central memory. Only a part is presumed to be available in central memory for processing at any time. This latter fact is reflected by using a window (buffer variable in PASCAL) on the file which makes only a part of the file (usually a single element) visible to the program at a time.

(2) Processing a file consists of alternating steps of (a) positioning the window on the file and (b) processing the element or elements visible within the window. Movement of the window to a new position is conceptually a separate step from the processing itself.

(3) A file has a lifetime (potentially) longer than that of the programs processing it, and hence the structure of the file is defined independently of the programs that access it.

(4) Implementation of file processing usually involves a limited form of concurrency. Typically two processes cooperate, one executing the user program and the other managing the buffering of blocks of data between secondary storage and buffers in central memory. From the buffer a local copy of the data visible in the processing window is provided to the program. As the window is moved, the buffer manager process determines when transfer of an entire block to secondary storage is necessary. The user is effectively

protected from managing these transfers himself, but the language concepts of "window" and "moving the window" reflect more abstractly the performance realities inherent in the implementation structure.

The model developed here for large array processing utilizes similar concepts. In this section the major features of the model are presented in a machine and language independent form. We state each of the major semantic aspects of the model, together with the rationale for its inclusion. In the next section, the model is presented more fully as an Ada "large array" abstract data type (package), which then provides both a syntax and an implementation model in terms of Ada semantics. The major features of the model are the following:

1. _Large array organization_. A "large array" is seen by the programmer as a single data object with the same logical organization as an ordinary array. That is, a one-dimensional large array is a linear sequence of homogeneous elements, a two-dimensional large array is a grid of rows and columns, and so forth.
Rationale. The programming burden is greatly simplified if the basic formulations of algorithms in terms of matrix algebra and other array processing structures can be retained without a major reorganization.

2. _Lifetimes of large arrays_. The lifetime of a "large array" ordinarily is different from that of a program which processes it. Thus its structure is defined independently of a program, and a program gains access to it through an OPEN operation similar to that used for

access to a file. A CLOSE operation terminates access to the array without destroying it.

Rationale. In applications, large arrays ordinarily represent data such as images or structural models that are constructed by other software systems or by other phases of a large program complex. They are often saved between processing phases and may be processed repeatedly by several different programs.

3. Windows. A "large array" is not visible to a single task as a unit at one time. Instead, a task needing to access it defines a "window" on it, where a window is a subarray of the whole. The elements visible in the window are available to the task for processing, treating the window elements as the elements of an ordinary (small) array.

Rationale. Restricting visibility to a window allows distributed storage of the array, with only the window elements present in the primary memory of the processor running the task that owns the window. The remainder of the array may reside on secondary storage, in other processor memories, or in other parts of a memory hierarchy.

4. Fixed-size windows. The size of a window is fixed when it is created and remains invariant throughout its lifetime.

Rationale. Dynamically changing window sizes (e.g., as found in OL/2 [14]) require substantial run-time overhead for referencing and storage management. Fixed size windows which can overlap array boundaries (see # 10 below) are simpler to implement, although somewhat less flexible.

5. Movement of windows. A window may be positioned at any arbitrary point on a large array (absolute movement) or moved from its current position to a new position defined in terms of its current position (relative movement), e.g., by moving a specified distance in a specified direction relative to the current position. Thus if the entire array is to be processed, a window is positioned at an initial position and then processing alternates with relative movement until the entire array has been traversed.

Rationale. "Window movement" represents abstractly the primary cost associated with large array processing, because moving a window ordinarily involves some transfer of data in and out of primary memory. The abstraction allows use of various blocking and buffering mechanisms in the implementation, as described in the following section. In addition, regular traversal patterns can be tied to iteration structures in the underlying language (e.g., the "iterators" of CLU [7]) with the implementation providing prefetching of the next block of data into a staging area during a previous iteration, thus overlapping processing in a window with the input-output required for a later iteration.

6. Access rights of windows. A window may be restricted to be "read-only", "write-only", or may be "read-write". A read-only window allows referencing but not assignment of values visible in the window. A write-only window allows assignment of new values only.

Rationale. Read-only windows provide different performance characteristics from read-write windows, because read-only windows require fewer data transfers and less mutual exclusion between tasks. In general, to maintain consistency of the data in a large array, only a single read-write window may be positioned over any given element, but multiple read-only windows over an element are acceptable. Write-only windows need not be initialized from secondary storage and require fewer data transfers when moved. Write-only windows are natural when extending an array or filling a newly created array.

7. Windows provide a local copy of the data. A window provides a local copy of the data visible in the large array. Updating of the actual large array values occurs only when the window is moved to a new position, or through an explicit WRITE operation, or when the window is CLOSE'd. An explicit READ operation allows updating the local copy of the large array values without window movement (e.g. when one task may be writing in an area that another is reading). Similarly the WRITE operation updates the large array with values from the local copy of the window without moving the window.

Rationale. For performance reasons again, we do not want updating of a window element via assignment to imply input-output to secondary storage. Instead only window movement (or explicit WRITE or CLOSE of a window) should cause input-output. This design allows efficient use of fairly large windows that minimize secondary storage traffic.

8. Multiple windows on different arrays. A task may have windows on several different arrays simultaneously and these windows may be moved asynchronously. A window may be opened or closed on an array at any time during execution.

Rationale. Many algorithms require simultaneous access to several arrays, e.g., to scan one while creating another.

9. _Multiple windows on the same array_. Several tasks may have windows concurrently on the same array. To preserve the basic integrity of the data as these windows move, a read-write or write-only window (an update window) cannot overlap other update windows. A read-only window is allowed to overlap with other types of windows. A task wishing to move an update window to a position overlapping the position of another update window must wait until the latter window is either moved or closed. The checking and waiting involved is part of the MOVE operation semantics and requires no special programmer action. _Rationale_. In working with large numbers of tasks moving windows asynchronously on the same array, it is relatively simple for the language run-time support programs to provide mutual exclusion and waiting when windows collide (as detailed in the next section). To require the programmer to do the checking explicitly would make programming much more difficult. Of course the mutual exclusion provided by the system raises the issues of deadlock and starvation, which must be addressed when implementation is considered.

10. _Windows and array boundaries_. A window may extend beyond the boundary of the array. At the time of creation of a read-only or read-write window, a default value for elements beyond the array boundary may be specified. The default value is returned for any reference to such an element. Assigning a value to an element outside the array boundary changes only the local copy of the window elements.

The array itself is not extended by such an assignment. When the window is moved, values outside the array boundary are lost. The single exception to this rule is the case of a linear array with a write-only window positioned just past the last element (greatest subscript). Assignments within such a window extend the array and are retained when the window is moved. An "end-of-structure" function allows a program to test whether a window extends over an array boundary.

Rationale. "Edge effects" when moving a window within a large array are a major source of programming problems. For example, when processing a symmetric matrix, one often wants to store only the upper triangular portion and move a window down the rows, starting with the element on the main diagonal. The rows get shorter as the window approaches the lower-right corner of the matrix, and it is convenient if the same window can be used, simply extending past the matrix boundary as needed. Similarly, if processing an image (matrix of pixels) with a fixed size window moving in fixed size increments, it is convenient if the window can overlap the image boundary at the end of each processing sweep.

Writing new values outside the array boundary has a simple semantics and implementation only in the case of extending a linear array. For a matrix, the semantics is complex if an extension of the array is intended, and the implementation is also troublesome. For simplicity, therefore, no dynamic extension of large arrays is allowed, except in the simple case. The "end-of-structure" function is the analogue of the usual "end-of-file" function for files and has similar uses.

11. Subwindows. A task may subdivide a window into smaller parts called subwindows. A subwindow is a subarray of a window. A subwindow may be passed as a parameter to a subtask (another task) for concurrent processing. However, a subwindow is semantically distinct from a window in three primary ways: (a) a subwindow cannot be moved, (b) assignment to a subwindow element implies immediate update of the corresponding window element, and (c) overlapping subwindows directly access the same element of the window (there are no local copies of subwindow elements). Processing using subwindows proceeds in distinct phases: (a) The main task positions its window appropriately on a large array, (b) subwindows are passed to subtasks for processing, and (c) when all subtasks have terminated (or paused) the main task moves its window to a new position and the cycle repeats.

Rationale. Processing using windows implies some secondary storage traffic and system overhead for mutual exclusion, particularly where several tasks are processing the same large array simultaneously. Subwindows allow several tasks to operate on the same part of a large array without major system overhead. Because subwindows cannot be moved independently, no secondary storage traffic is involved. Because no local copies of array elements are maintained and assignment causes immediate update of an element in all subwindows within which it is visible, the programmer is responsible for mutual exclusion and consistency of the data, rather than the system. For many applications, this alternative is appropriate. For example, a window may be positioned over several dozen rows of a large array. A subtask may be initiated to process each row, with a subwindow on the appropriate row. Since these subwindows can never overlap, consistency within the window is not an issue, and the system overhead for checking is not required. Also, secondary storage traffic is greatly reduced by moving the large window as a unit rather than each "row" subwindow individually. At the same time, system checking and mutual exclusion on movement of the larger window allows other tasks in the system to process the same large array for other purposes concurrently. Shared memory architectures allow subwindows to be implemented as simple descriptors (dope vectors) with essentially no overhead in their concurrent processing. In non-shared memory architectures, references and assignments to subwindows require communication to the task owning the window, which may substantially increase the cost of use. For this reason, subwindows are particularly attractive for use where tasks have shared memory available.

Examples. Several short examples are presented here, with a more extended example in a following section. Since we have no syntax for programming these examples, only an intuitive sketch of the semantics is provided.

Example 1. Ordinary sequential file processing. Sequential file processing is the simplest case of large array processing. A task positions a read-only window at the start of a large linear array (to read the array file). The window is the equivalent of the PASCAL "buffer variable". Processing the element(s) within the window alternates with advancing the window until the entire array has been traversed. To write the array, a write-only window is positioned one element beyond the end, and assignment to the window element alternates with forward movement until all data has been written. Note that the large array semantics allows the usual implementation structures for file processing to be adapted to this larger conceptual model for array processing.

Example 2. UNIX[2]-like pipes. A pipe in UNIX-like systems is a file that is written by one task while a second simultaneously reads and processes the newly written elements. The second task may lag arbitrarily far behind the first in its processing. The large array model allows this structure to be represented by the first task having a write-only window at the end of the array-file (as in Example 1 above) and the second task having a read-write window positioned on some existing part of the array and advancing toward the first window. Note however that the array is not "consumed" by the second task reading it, so several tasks may have windows that allow them to read and process the array concurrently.

Example 3. Factoring a banded symmetric matrix. In solving a large system of equations in matrix form:

$$A X = B$$

the matrix is often symmetric and "banded", meaning each row has only a

[2] UNIX is a trademark of Bell Laboratories.

maximum of some constant "beta" non-zero elements, starting with the element on the main diagonal. "Beta" is termed the "bandwidth" of the matrix. Only the elements within the non-zero band are stored, as an N x "beta" matrix if the original matrix was N x N. A factoring algorithm typically might work down the band, instantiating "beta"+1 tasks, each of which works on one row of the matrix at a time. The first task takes the first row, the "pivot row", and sends it to the other tasks. Each of these tasks uses the pivot row to perform calculations on the row it is currently assigned. Figure 1 illustrates this processing structure.

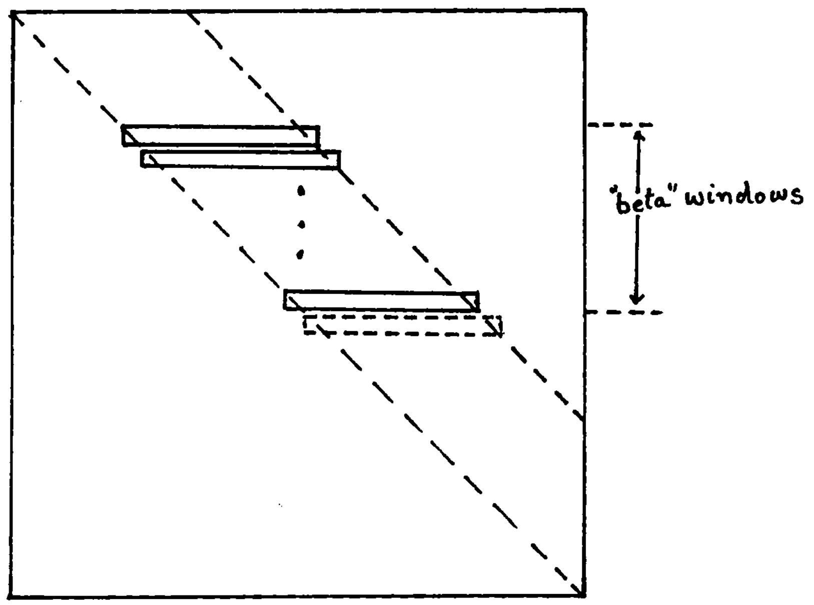

Figure 1. Solution of a banded matrix

Only "beta"+1 tasks are needed because only the rows with at least one non-zero element in the same column as a non-zero of the pivot row need to be processed in each step.

After sending the pivot row to all other tasks, the first task moves its window down to the row "beta"+1 rows below its current row. The second task then has the new pivot row, and the process repeats. The tasks progressively move their windows down the band until the entire array is processed.

4. An Implementation Model

In the last section an overview of the language concepts needed for concurrent processing of large arrays has been provided. In this section an implementation model for these concepts is sketched briefly. An actual implementation as an Ada package is presented in the next section.

The implementation model for large arrays parallels to some extent the implementation of file processing on sequential machines. As noted before, for sequential file processing, a buffering process manages the transfer of blocks of file components between the secondary and primary storage, copying the components of the file into and from the buffer variable associated with the file as needed. The concept of windows on a large array, as seen in the last section, already embodies the idea of a private copy of the elements of the large array visible through the windows.

The large array itself is divided into subarrays or blocks for implementation purposes as shown in Figure 2. The I/O of the large array is performed in terms of the blocks, i.e., the large array is stored on the secondary storage as blocks and the blocks are transferred in and out of primary storage as needed.

A window, when stationed on the large array, may fall completely within a block or may overlap several blocks (see Figure 2). To

---

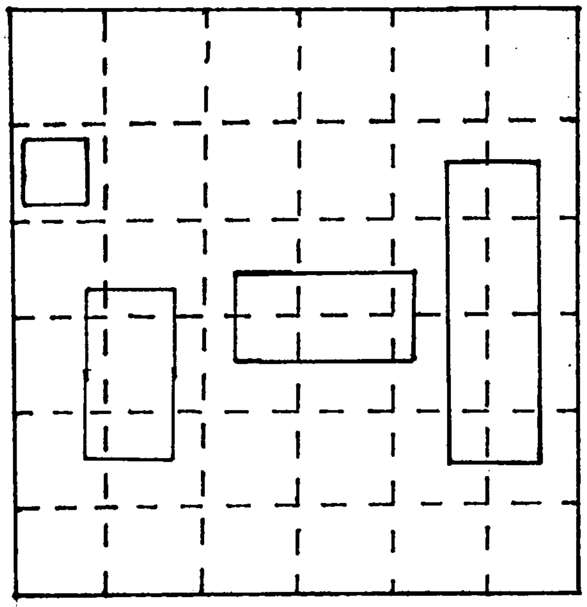

Figure 2. A large array divided into blocks with four windows on it.

---- indicates block boundaries
____ indicates window boundaries.

---

provide a copy of the elements, the appropriate portions of the blocks covered by the window are transferred sequentially to the window. When the window is moved, the old values are written back. This implies that each of the blocks overlapped by the window in the old position must be updated.

The division of large arrays into blocks also supports concurrency. As long as two windows are stationed on the large array so that they cover totally separate blocks, all processing required for the movements of the windows can be performed in parallel. Such a situation is depicted by the two windows on the left in Figure 2. On the other hand, if the windows are stationed so that they overlap common blocks (the two windows on the right in Figure 2), then access to the common blocks has to be serialized so as to preserve the integrity of the data. The tasks owning the windows can still process the window copies of the elements in parallel, with the serialization occurring only during the reading and writing process between the window and the actual elements of the large array.

A block of the large array is required to be resident in the primary memory only if there is a window covering it. Thus a simple demand paging policy can be implemented wherein a block is paged in from secondary storage only when a window moves onto it. The block can be replaced when the window moves on and no other window has covered it in the meantime.

The model presented above attempts to find a solution to the two related problems in the concurrent processing of large arrays as discussed at the beginning of the section. Windows provide a means of specifying the elements of a large array needed by the task for further processing without the programmer having to code the details of data movement between the primary and secondary storage. Tasks can use independent windows stationed on the same large array to process the array in parallel. The windows mutually exclude each other from the same portion of the array (when required). This provides for the maximum possible concurrent activity with no window unnecessarily inhibiting the processing in another window when such mutual exclusion is not needed. The model allows the language structure to reflect the performance realities of distributed storage and processing without unduly burdening the user with implementation details.

5. An Ada Implementation of Large Arrays

The model for processing large arrays in parallel needs to be a part of a system which provides other primitives necessary for distributed processing such as creation of independent tasks, communication between tasks, etc. To provide a precise semantics (and syntax) for the concepts of the model, we can consider "large array" as a new abstract data type. Although Ada does not provide any higher level constructs for the distributed control of data structures, it does provide a generic package facility which may be used to define this new abstract data type. The Ada tasking facility provides a "virtual computer" that may be considered as an abstract distributed machine. Within this machine, the large array package defines a detailed implementation of the model for processing of large arrays in parallel. The package consists of all the necessary type definitions for large arrays, windows and subwindows along with procedure and function definitions for the relevant operations on the above defined types. Such a package definition provides a clear and precise syntax (although not entirely ideal) for creating and operating on large arrays in algorithms designed for concurrent processing of arrays.

Since the large array package is coded in Ada, a working knowledge of the language is assumed for the purposes of describing the package. The large array package has been implemented on a VAX 11/780 using the UNIX implementation of the NYU/Ada ED translator and interpreter.

5.1. The Large Array Package

In this section the basic structure of the package is described. Most of the scientific applications involving large arrays generally require only two-dimensional arrays. Hence the package, as presented here, supports only two-dimensional large arrays. The extension of the package to higher dimensioned arrays is straightforward. The public part of the package has been provided in Appendix A. For a detailed listing of the package refer to [10].

5.1.1. Large Arrays

The generic package *LARGE_ARRAY_PKGE* can be instantiated with any predefined or user defined type being supplied as actual parameter to match the formal parameter *ELEMENT*. The following statements declare *FLOAT_ARRAY* as a new instantiation of the generic package to support arrays with elements of type *FLOAT*:

```
 package FLOAT_ARRAY is new LARGE_ARRAY_PKGE(FLOAT);
use FLOAT_ARRAY;
```

Preexisting arrays in external files can be accessed by attaching one of these large arrays via the procedure *OPEN_LARGE_ARRAY* while new arrays can be created using the procedure *CREATE_LARGE_ARRAY*. External files are named by a character string. The row and column bounds of a large array are specified at the time of associating the large array with an external file and remain fixed during the existence of the arrays. For example

```
 A, X : LARGE_ARRAY;

OPEN_LARGE_ARRAY ( A,
                   row_low_bd  => 1,
                   row_high_bd => n,
                   col_low_bd  => 1,
                   col_high_bd => n,
                   name        => "A_file");[3]

CREATE_LARGE_ARRAY ( X,
                     row_low_bd  => 1,
                     row_high_bd => n,
                     col_low_bd  => 1,
                     col_high_bd => m,
                     name        => "X_file");
```

[3] Ada allows parameters to a particular subprogram either as a list, which associates actual and formal parameters by position, or by preceding the actual parameter by "formal_name =>" where formal_name is the name of the associated formal parameter. Explicitly named parameters are used for clarity in many of these examples, but positional parameters are used where no confusion is likely.

A and *X* are declared as *LARGE_ARRAYs*. The large array *A* is associated with an already existing external file *A_file* and is specified to be an n by n array while a file *X_file* is created to be associated with the n by m large array *X*. Procedure *CLOSE_LARGE_ARRAY* can be used to sever the association of an internal large array with its associated external file while procedure *DELETE_LARGE_ARRAY* deletes the associated external file:

Internally a large array is viewed as a sequence of blocks, each block being defined as a square subarray of the large array as shown in Figure 2. The blocks are numbered sequentially starting from the top left corner and increasing along the columns and then rows.

The row and column bounds are used to determine the total number of blocks needed for the large array. For each of the blocks constituting the large array a "monitor"-like task - *BLOCK_CONTROLLER* - is initiated to control access to the block. The reading in and writing out of the blocks to secondary storage is performed by the associated task as and when required. Note that the *BLOCK_CONTROLLER* tasks are internal to the package and are not visible to the user.

5.1.2. Windows

Windows are attached to a particular large array using the procedure *CREATE_WINDOW*. The same procedure is also used to specify the size of the window and its privileges, i.e., whether it is read-only(R), write-only(W) or read-write(RW). The row and column increments to be used for relative movement of the window are also passed as parameters to the procedure. The user can specify an edge_element which will be used to fill out the portion of the window which does not lie within the bounds of the associated large array when the window is moved past the edge of the array. The statements

```
A_window, X_window : WINDOW;
CREATE_WINDOW( A_window,
               row_size => n, col_size => n,
               inmode   => R,
               row_inc  => 0, col_inc  => 0,
               ar       => A,
               edge     => FALSE, edge_element => 0.0);

CREATE_WINDOW( X_window,
               row_size => n, col_size => 1,
               inmode   => W,
               row_inc  => 0, col_inc  => 1,
               ar       => X,
               edge     => TRUE, edge_element => 1.0);
```

declare two windows *A_window* and *X_window*. The window *A_window* is associated with the large array *A*. It is a read-only window and its size is n by n. The move increments are set to zero, implying that it will not be moved via relative movements. The *X_window* is a n by 1 write-only window associated with the large array *X*. The move increments are (0,1), i.e., each relative move will displace it one column to the right on the large array. An *edge_element* of 1.0 is specified so that if the window is positioned such that it lies partially outside the large array, the elements of the window not overlapping the array will be set to 1.0. The exception *NONEXISTENT_ARRAY_ERROR* is raised if the large array on which window is to be stationed has not yet been created or opened.

With each variable of type *WINDOW*, the following data structures are visible to the task declaring the variable:

```
 type WINDOW is access WINDOW_DESC;

type WINDOW_DESC is
 record
   win     : MATRIX_ACCESS;
   info    : WINDOW_INFO;
end record;
```

Thus a window is viewed by the user as a local array of specified size (the component *win*) along with information which is private to the package (the component *info*) . The processing of the elements of the large array visible through the window is hence performed in a manner analogous to the processing of an ordinary small array. That is to say, the accessing of window elements is done through subscripting the component *win* relative to the origin of the window rather than the origin of the large array. In the following statement, the value of the (2,3)th element of the *A_window* is assigned to the (4,1)th element of *X_window*:

```
X_window.win(4,1) := A_window.win(2,3);
```

The position of the windows on the large arrays determines the actual elements of the large arrays used in the above assignment statement. Thus if the *X_window* is positioned at the fifth column of the large array *X* then *X(4,5)* is actually assigned.

Window movement

Two methods of window movement are provided: absolute and relative. The *SET* procedure moves the window to the indicated absolute position on the array. Thus *SET* can be used to establish the initial position of the window on the array. In the following statements, the *A_window* is set at position (1,1) on the large array *A* so as to cover the whole array while *X_window* is set at position (1,5) so as to cover the 5th column of the large array *X*:

```
SET(A_window, new_row => 1, new_col => 1);
SET(X_window, new_row => 1, new_col => 5);
```

The *MOVE* procedure, on the other hand, uses the row and column increments defined while creating the window, to move to a new position relative to the present position. The procedure *MOVE* just calls *SET* with the new absolute position. The statement

```
MOVE (X);
```

uses the move increments specified during the creation of *X_window* to move the window one column to the right, i.e., the 6th column of large array *X*.

Whenever a window is moved, a copy of the elements visible through the window is provided in the local array *win* (component of *WINDOW_DESC*) . When the window is moved again, the values of the elements are written back to the external large array before the move is executed. Thus each task processes its private copy of the elements visible through the window. The actual array is updated <u>only</u> when the window is moved away. The reading and writing of the window elements is obviously dependent on the privileges of the window, i.e., no writing is done for read-only windows while no reading is performed for write-only windows.

Procedure *READ* can be used to copy the large array elements covered by the window into the local array of the window without actually moving the window. Similarly procedure *WRITE* is available to update the large array from the local array without moving the window.

Depending upon the size of the window and the position of the window within the large array, the window may partially or fully cover one or more blocks of the array (see Figure 2). Internally for each window a list of blocks covered by the window is maintained as *block_list*. The procedure *READ* makes an entry call to each of the tasks associated with the blocks in this list sequentially, to update the appropriate portions of the window. Similarly, *WRITE* makes an entry call to the tasks for updating the appropriate portions of the blocks.

When a window is to be moved to a new position, several steps are performed by the procedure *SET*. First the elements at the present position are updated (for update windows only) by calling the *WRITE* procedure. The procedure *RELEASE_BLOCKS* is then called to detach the window from the blocks it was covering. *RELEASE_BLOCKS* makes entry calls to the tasks controlling the blocks covered by the window to detach the window from the blocks. Note procedures *WRITE* and *RELEASE_BLOCKS* are called only if the *block_list* is nonempty. An empty *block_list* indicates that either the window has not yet been initially stationed on the array or that its position is such that it is totally outside the array and hence is not covering any of the blocks of the array.

Next the blocks that would be covered by the window in the new position are determined. *SET* calls the procedure *OBTAIN_BLOCKS* for this purpose. *OBTAIN_BLOCKS* determines the position of the window with respect to the large array and the blocks covered by the window at this position. Entry calls are made to the tasks controlling these blocks to "attach" the window to the blocks. If another incompatible window covers the same elements, the "attach" call is delayed until the other window is moved. Once attach entry calls for all the blocks covered by the window have been successfully completed, the elements at the new position are read into the window via the procedure *READ* (for read-only and read-write windows only), thus completing the operation of moving a window to a new position. The procedures *RELEASE_BLOCKS* and *OBTAIN_BLOCKS* are not visible to the programs using the package.

5.1.3. Subwindows

Subwindows can be overlayed on a window via the procedure *CREATE_SUBWINDOW* provided for this purpose. The window with which the subwindow is to be associated, its size, and its position within the window are passed as parameters. The following statements

```
 A_sw  : array(1..n) of SUBWINDOW;
X_sw  : SUBWINDOW;

for i in 1..n
 loop
    CREATE_SUBWINDOW(A_sw(i),
                        row_size => n, col_size => 1,
                        row_pos  => 1, col_pos  => i,
                        wind     => A_window         );
end loop;

CREATE_SUBWINDOW(X_sw,
                    row_size => n/2, col_size => 1,
                    row_pos  => 1,   col_pos  => 1,
                    wind     => X_window           );
```

declare a subwindow *X_sw* and an array of subwindows *A_sw*. Each of the *A_sw* subwindows is created to be a n by 1 subwindow associated with the window *A_window*. The i'th subwindow *A_sw(i)* is overlayed on the i'th column of the *A_window*. The *X_sw* subwindow is created so as to cover the top n/2 elements of the *X_window*.

The subwindow should lie entirely within the window with which it is associated, otherwise the exception *SUBWINDOW_OUTSIDE_WINDOW* is raised. For each window a list of subwindows overlaying it is maintained. On creation the subwindow is attached to the end of the *subwindow_list* for the associated window.

Procedure *ASSIGN* can be used to assign a value to a particular element of the subwindow. Function *GET* returns the value of the specified element. For example, here

ASSIGN(X_sw, 4, 1, GET(A_sw(5), 1, 1));

results in the first element of the 5th *A_sw* subwindow being assigned to the 4th element of *X_sw*. These two subprograms work directly on the window to which the subwindow is attached since a subwindow does not have its own private copy of the elements.

Note that there are no procedures for moving subwindows. Only the window on which the subwindow is overlayed can be moved. Thus when a window is moved, conceptually all the subwindows attached to it are automatically moved while preserving their relative positions.

5.1.4. End of Structure Functions

The functions *EOS* can be used to determine if a window or subwindow has reached the end of the structure, i.e., the edge of the large array on which the window is stationed. A window can be totally inside the large array or totally outside it. The window can also be straddling the large array along one of the edges or one of the corners. The functions *EOS* returns a value of the enumeration type *DIRECTIONS*. A value *INSIDE* is returned if the window or subwindow is totally inside the large array while *OUTSIDE* is returned if the window or subwindow is totally outside the large array. The compass directions are returned for the other eight possible positions of the window or subwindow with respect to the large array. Thus *N* is returned when the top edge is straddled, *NW* is returned if the top left hand corner is straddled and so on.

The *LARGE_ARRAY_PACKAGE* also provides various other functions to determine the properties of large arrays, windows and subwindows. The package has been described in greater detail in [10].

5.2. Mutual Exclusion and Deadlock

The semantics of the Ada tasking facilities have been used to provide mutual exclusion to tasks wishing to access the same block of the large array simultaneously. Since an independent *BLOCK_CONTROLLER* task controls each of the blocks, the *BLOCK_CONTROLLER* task performs the role of a "monitor" for the block. As long as two tasks are moving their windows over different parts of the large array, the movement can be performed in parallel. It is only when the two windows have to move onto the same block that their movements are carried out in a sequential manner. Also the reading and writing can be performed by only one task at a time. This does not inhibit parallel activity, since once a copy of the relevant portion of the block has been made into a window, independent tasks can process their windows concurrently.

Windows can be moved asynchronously over a large array by the tasks owning the windows. In moving the windows, the tasks are requesting "resources", i.e., making attach entry calls to the tasks controlling the blocks of the large array in an independent and asynchronous manner. A task can be blocked if the movement of the window would cause an incompatible overlap with another window already stationed on the block. Thus a deadlock could potentially occur if two windows each simultaneously request to be attached to blocks on which the other is already stationed in such a fashion as to cause an incompatible overlap in both blocks. Such a deadlock is avoided by a simple resource ordering strategy. The blocks of a large array are numbered sequentially and must be requested by each task in a fixed sequence. A *SET* operation (and hence the *MOVE* operation) proceeds by first releasing the blocks covered by the window in the old position and then attaching the window to the blocks in its new position in the order of the block numbers. No reading or writing of the window elements at the new position may occur until the window has been successfully attached to all the blocks that it has to cover. This ensures that two windows cannot each be vying for a block already covered by the other window, thus avoiding deadlock.

5.3. Block size

Each large array is internally divided into blocks. The size of the blocks is dependent on several competing factors. Since the block is used as a unit for transfer of data between secondary storage and primary memory(s), the optimum size for I/O transfer influences the block size. Each block is controlled by an independent task. Thus if the block size is small, the number of tasks controlling the blocks will become large. On the other hand, if the block size is large then the concurrency available during the movement of windows is reduced. The trade off between the amount of parallelism and the number of tasks has to be weighed in conjunction with the optimum I/O transfer size when determining the size of the blocks of the large arrays.

6. Array Processing: An Example

An algorithm for solving a triangular system of equations is elaborated in this section. The equations are represented by the following matrix equation:

$$Ax = b$$

where the matrix A is an n by n upper triangular matrix. The algorithm, coded in Ada (see Appendix B), essentially uses a direct back solve method for solving the equations for a set of m right hand side vectors b.

The algorithm utilizes n subtasks for solving the system of equations - n-1 *BACK_SOLVE* subtasks and one *MAIN_BACK_SOLVE*. The n subtasks are conceptually arranged in a linear chain, each having a right and left neighbor (except of course the subtasks at the end points). The subtask *MAIN_BACK_SOLVE* is the n'th subtask in the chain.

The algorithm is set up so that each subtask views one column of the array *A* and calculates one element of the solution vector x. The subtask *MAIN_BACK_SOLVE* "receives" the right hand side vector b and calculates the n'th element of x using the n'th element of b and the A(n,n) element. It also calculates the contribution of the n'th column of the array A to the solution vector and passes on the partially calculated x vector to its left neighbor. The id'th subtask BACK_SOLVE receives the partially calculated x vector from its right neighbor, calculates the id'th element of the x vector and passes on the partially calculated values of x, updated by contribution of the id'th column of the array A, to its left neighbor.

The procedure *MAIN* declares *A*, *X* and *B* as *LARGE_ARRAYs* and opens *A* and *B*, associating them with already existing external files *A_file* and *B_file* respectively. It also creates a new file *X_file*, to be associated with the *LARGE_ARRAY* *X*. A read-only window *A_window*, is also defined by *MAIN* and associated with the *LARGE_ARRAY* *A*. The window *A_window* is as large as the array itself and is positioned so as to cover the whole array.

Each of the *BACK_SOLVE* subtasks declares a subwindow *A_sw* on the window *A_window* and overlays it such that the subwindow for subtask *id* is at the id'th column of the window (see Figure 3(a)). Also a one element write-only window is declared by each subtask on the *LARGE_ARRAY* *X* and initially positioned on the first column of the id'th row.

The subtask *MAIN_BACK_SOLVE*, in addition to the *A_sw* subwindow and the *X_window* also has a window, *B_window*, on the *LARGE_ARRAY* *B* (Figure 3(c)). This window is initially positioned at the first column of the array *B*. After setting the *B_window* on the first column of the *B* array, the subtask calculates the n'th element of the x vector and writes it in the *X_window*. It then passes the partially calculated n-1 values to its left neighbor, subtask n-1, and then moves the *B_window* and *X_window* one position to the right on the respective arrays so as to process the next b vector (Figure 3(b) &

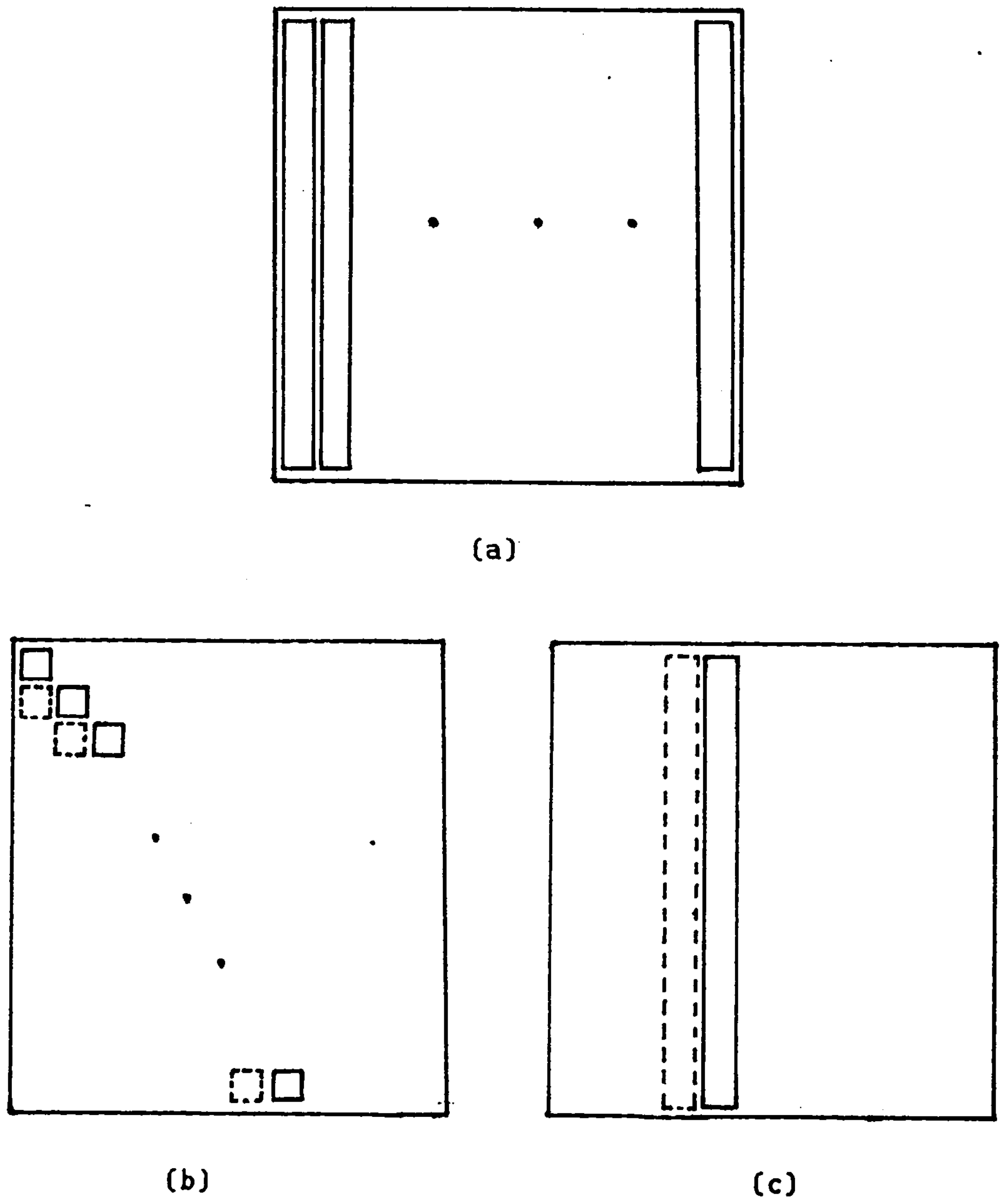

Figure 3. Windows and subwindows for the back solve process
(a) Window and subwindows on large array A
(b) Movement of windows on large array X
(c) Movement of the window on large array B

3(c)). Note that the move increments for both the windows are appropriately set when defining the windows. Processing is continued until the end of structure is reached (*EOS(B_WINDOW) = OUTSIDE*) .

The id'th subtask *BACK_SOLVE* receives id elements of the partially calculated x values from its right neighbor and generates the id'th element of the x vector in the *X_window*. It communicates partially calculated x values to its left neighbor and then moves its *X_window* (Figure 3(b)). This is repeated until all the right hand sides have been processed. For each right hand side vector, the id'th subtask produces the id'th element of the corresponding x vector. Thus the end of processing is signified by the fact that the *X_window* has been moved outside the *X* array (*EOS(X_window) = OUTSIDE*) . When all subtasks have finished processing the main program closes all the large_arrays.

As noted before, the above algorithm has been described without assuming an underlying architecture. An implementation of the algorithm on the NASA Finite Element Machine is described in [9].

7. Conclusion

The minimal support for large data structures in proposed languages for multiprocessor architectures has led to the present investigation. The basic aim of the conceptual model presented in this paper, is to lay the groundwork for incorporating the concept of large arrays into parallel languages. The model provides the user with an easy means of specifying not only the distributed storage but also the distributed processing of large arrays in a high level language without resorting to low level coding within his program.

The basic data structure "large array" is the same as that of an ordinary small array, but it is clear that the manner in which a large array is to be processed in a multiprocessor environment cannot be the same because of the performance realities involved in its processing. A similar dichotomy can be seen in sequential languages where the two concepts of files and vectors are not merged even though files can be regarded as vectors of records.

The conceptual model for parallel processing of large arrays, as presented in this paper, is an attempt to fill the gap between what is provided by the present day parallel languages and what is needed by the users of such languages. The central concepts of the model, windows and subwindows, provide the means for representing both the distributed storage and concurrent processing of large arrays in programs in a way that is natural for array processing.

The Ada large array package was implemented so as to specify the semantics of the above concepts in more concrete terms. Various factors dictated the choice of Ada as a vehicle for the specification of the semantics. Ada provides excellent facilities for implementing abstract data types through generic packages. The Ada tasking facilities can be used to represent a virtual abstract multiprocessor system. Since the thrust of this research was the study of large arrays in a parallel environment rather than the use of concurrent systems per se, Ada tasking facilities provided a base on which the large array model could be implemented.

The Ada package provides a particular implementation of the general model for parallel processing of large arrays without any assumptions about the underlying architecture. However the performance realities associated with each particular architecture have to be taken into account when considering the implementation of the model on a particular machine. The model provides a general framework which can be tailored to fit the nuances of a particular architecture. The strategies for implementing the model on four architectures namely: the NASA MPP [3], the Intel 432 system [1], the NASA Finite Element Machine [18], and the University of Maryland ZMOB [15] have been described in [10].

8. REFERENCES

1. "Introduction to the iAPX 432 Architecture," Intel Corporation (1981).

2. "Reference Manual for the ADA Language," U. S. Department of Defense, Washington, D.C. (July 1982).

3. Batcher, K. E. "Design of a Massively Parallel Processor," *IEEE Transactions on Computer C-29,* 9 (Sept. 1980), pp. 836-840.

4. Gajski, D., Kuck, D., Lawrie, D., and Sameh, A. "Cedar - A Large Scale Multiprocessor," *Proceedings of the 1983 International Conference on Parallel Processing* (August 1983), pp. 524-529.

5. Hoare, C. A. R. "Communicating Sequential Processes," *Communications of the ACM 21,* 8 (August 1978), pp. 666-677.

6. Jones, A. and Schwarz, P. "Experience Using Multiprocessor Architectures - A Status Report," *ACM Computing Surveys 12,* 3 (June 1980), pp. 121-166.

7. Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. "Abstraction Mechanism in CLU," *Communications of the ACM 20,* 8 (August 1977), pp. 564-576.

8. Liskov, B. and Scheifler, R. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *Ninth ACM Symposium on Principles of Programming Languages* (January 1982), pp. 7-

19, Albuquerque, NM.

9. Mehrotra, P. and Pratt, T. W. "Language Concepts for Distributed Processing of Large Arrays," *Proceedings ACM Symposium on Principles of Distributed Computing* (August 1982), , Ottawa, Canada

10. Mehrotra, P. "Parallel Computation on Large Arrays," *Ph.D. Thesis* (August 1982), University of Virginia, Charlottesville, Va.

11. Millstein, R. E. "Control Structures in ILLIAC IV FORTRAN," *Communications of the ACM* *16,*1 0 (October 1973), pp. 621-627.

12. Perrott, R. H. "A Language for Array and Vector Processors," *ACM Transactions on Programming Languages and Systems* *1,*2 (October 1979), pp. 177-195.

13. Perrott, R. H. and Stevenson, D. K. "User's Experience with the ILLIAC IV System and its Programming Languages," *Sigplan Notices* *16,7* (July 1981), pp. 75-88.

14. Phillips, J. R. and Adams, H. C. "Dynamic Partitioning for Array Languages," *Communications of the Association for Computing Machinery* *15,*1 2 (December 1972), pp. 1023-1032.

15. Rieger, C. "ZMOB: A Mob of 256 Cooperative Z80-based Microprocessors," *Proceedings Image Understanding Conference, Los Angeles, CA.* (1979), pp. 25-30.

16. Sejnowski, M. C., Upchurch, E. T., Kapur, R. N., Charlu, D. P. S., and Lipovski, G. J. "An Overview of the Texas Reconfigurable Array Computer," *AFIPS Conference Proceedings 1980 NCC* (1980), pp. 631-641.

17. Snyder, L. "Introduction to the Configurable, Highly Parallel Computer," *Computer 15*, 1 (January 1982), pp. 47-56.

18. Storaasli, O. O., Peebles, S. W., Crockett, T. W., Knott, J. D., and Adams, L. "The Finite Element Machine: An Experiment in Parallel Processing," *Research in Structural and Solid Mechanics, NASA Conference Publication 2245* (October 1982), pp. 201-217, Washington, D.C.

19. Uhr, L. "A Language for Parallel Processing of Arrays, Embedded in PASCAL," TR-365, Computer Science Department, University of Wisconsin-Madison (September 1979).

APPENDIX A

Ada Generic LARGE_ARRAY Package

```
--------------------------------------------------------------
--                                                          --
--   The following generic package, LARGE_ARRAY_PKGE,       --
--   can be used for parallel processing of large arrays.   --
--   The package defines the needed data types and          --
--   subprograms for such processing.                       --
--   The package can be instantiated with any actual type   --
--   corresponding to the formal parameter ELEMENT.         --
--                                                          --
--------------------------------------------------------------

generic

    type ELEMENT is  private;

  package LARGE_ARRAY_PKGE is

    type LARGE_ARRAY is private;

    type MATRIX is array(INTEGER range <>, INTEGER range <>)
                                        of ELEMENT;
    type MATRIX_ACCESS is access MATRIX;

    type WINDOW_INFO is private;

    type WINDOW_DESC is
     record
       win     : MATRIX_ACCESS;
       info    : WINDOW_INFO;
     end record;

    type WINDOW is access WINDOW_DESC;

    type SUBWINDOW is private;

    type PRIVILEGES is (R,W,RW);

    type DIRECTIONS is (N,S,E,W,NE,NW,SE,SW,INSIDE,OUTSIDE);
```

```
-- Procedures to create or open large arrays

   procedure CREATE_LARGE_ARRAY( ar   : in out LARGE_ARRAY;
                                 row_low_bd,
                                 row_high_bd,
                                 col_low_bd,
                                 col_high_bd : in INTEGER;
                                 name        : in STRING);

   procedure OPEN_LARGE_ARRAY  ( ar   : in out LARGE_ARRAY;
                                 row_low_bd,
                                 row_high_bd,
                                 col_low_bd,
                                 col_high_bd : in INTEGER;
                                 name        : in STRING);


-- Procedures to delete or close large arrays

   procedure DELETE_LARGE_ARRAY( ar: in out LARGE_ARRAY);

   procedure CLOSE_LARGE_ARRAY ( ar: in out LARGE_ARRAY);


-- Procedures to create windows and subwindows

   procedure CREATE_WINDOW( wind                 : in out WINDOW;
                            row_size, col_size : in NATURAL;
                            inmode               : in PRIVILEGES;
                            row_inc, col_inc     : in INTEGER;
                            ar                   : in LARGE_ARRAY;
                            edge                 : in BOOLEAN;
                            edge_element         : in ELEMENT);

   procedure CREATE_SUBWINDOW( subwin           : in out SUBWINDOW;
                               row_size, col_size,
                               row_pos, col_pos   : in NATURAL;
                               inmode             : in PRIVILEGES;
                               wind               : in WINDOW);
```

```
-- Procedures to move windows.
--  Note that a move implies
--    writing the last position (if the window is not read-only)
--    reading the new position (if the window is not write-only).

   procedure MOVE( wind    : in out WINDOW); -- relative movement

   procedure SET ( wind    : in out WINDOW;
                   new_row,
                   new_col : in INTEGER   ); -- absolute movement

-- Procedures to read and write windows without moving them.

   procedure READ ( wind: in out WINDOW);
   procedure WRITE( wind: in     WINDOW);

-- Subprograms to assign and retrieve values of specified
--  elements of subwindows.

   procedure ASSIGN (subwin         : in out SUBWINDOW;
                     row, col       : in NATURAL;
                     value          : in ELEMENT);
   function  GET ( subwin           : in SUBWINDOW;
                   row, col         : in NATURAL) return ELEMENT;

-- Functions to determine the end of structure.

   function EOS(wind: in    WINDOW) return DIRECTIONS;
   function EOS(wind: in SUBWINDOW) return DIRECTIONS;


-- Various functions to determine properties of large arrays,
--     windows and subwindows.


-- Exceptions that can be raised in the package.

   ARRAY_SIZE_ERROR                  : exception;
   ARRAY_DIFFERENT_ERROR             : exception;
   READ_WRITE_MODE_ERROR             : exception;
   NONEXISTENT_ARRAY_ERROR           : exception;
   NONEXISTENT_WINDOW_ERROR          : exception;
   SUBWINDOW_OUTSIDE_WINDOW          : exception;

end LARGE_ARRAY_PKGE;
```

APPENDIX B

Ada Program for the Back Solve Process

```
with LARGE_ARRAY_PKGE;
procedure MAIN is

    package FLOAT_ARRAY is new LARGE_ARRAY_PKGE(FLOAT);
    use FLOAT_ARRAY;

    n         : constant NATURAL ; -- Size of the A matrix
    m         : constant NATURAL ; -- Number of right hand sides

    A,X,B     : LARGE_ARRAY;
    A_window  : WINDOW;

    begin

         OPEN_LARGE_ARRAY( A, row_low_bd  => 1, row_high_bd => n,
                              col_low_bd  => 1, col_high_bd => n,
                              name        => "A_file");

         OPEN_LARGE_ARRAY( B, row_low_bd  => 1, row_high_bd => n,
                              col_low_bd  => 1, col_high_bd => m,
                              name        => "B_file");

         CREATE_LARGE_ARRAY( X, row_low_bd  => 1, row_high_bd => n,
                             col_low_bd  => 1, col_high_bd => m,
                             name        => "X_file");

         CREATE_WINDOW( A_window, row_size => n, col_size => n,
                                  inmode   => R,
                                  row_inc  => 0, col_inc  => 0,
                                  ar       => A,
                                  edge     => FALSE, edge_element => 0.0);
         SET(A_window, new_row => 1, new_col => 1);

        declare

            type VECTOR is array(1..n) of FLOAT;

            task type BACK_SOLVE is
               entry WHO_AM_I(self_id : NATURAL);
               entry NEXT( x : VECTOR);
            end BACK_SOLVE;

            task MAIN_BACK_SOLVE is
            end MAIN_BACK_SOLVE;

            solve   : array(1..n-1) of BACK_SOLVE;
```

```
task body MAIN_BACK_SOLVE is

   B_window, X_window  : WINDOW;
   A_sw                : SUBWINDOW;
   partial_x           : VECTOR;

 begin

  CREATE_SUBWINDOW(A_sw, row_size => n, col_size => 1,
                         row_pos  => 1, col_pos  => n,
                         wind     => A_window         );

  CREATE_WINDOW(X_window, row_size => 1, col_size => 1,
                         inmode   => W,
                         row_inc  => 0, col_inc  => 1,
                         ar       => X,
                         edge     => FALSE, edge_element => 0.0);
  SET(X_window, new_row => n, new_col => 1);

  CREATE_WINDOW(B_window, row_size => n, col_size => 1,
                         inmode   => R,
                         row_inc  => 0, col_inc  => 1,
                         ar       => B,
                         edge     => FALSE, edge_element => 0.0);
  SET(B_window, new_row => 1, new_col => 1);

 solve_cycle:
  loop

    X_window.win(1,1) := B_window.win(n,1) / GET(A_sw,n,1);

    for i in 1..n-1
     loop
      partial_x(i) := (B_window.win(i,1)
                    - GET(A_sw,i,1) * X_window.win(1,1));
    end loop;

    solve(n-1).NEXT(partial_x);

    MOVE(B_window);
    MOVE(X_window);

    exit solve_cycle when (EOS(B_window) = OUTSIDE);

  end loop solve_cycle;

end MAIN_BACK_SOLVE;
```

```
task body BACK_SOLVE is

    X_window     : WINDOW;
    A_sw         : SUBWINDOW;
    id           : NATURAL;
    partial_x    : VECTOR;

begin

  accept WHO_AM_I ( self_id : NATURAL) do
     id := self_id;
  end WHO_AM_I;

  CREATE_SUBWINDOW(A_sw, row_size => n, col_size => 1,
                         row_pos  => 1, col_pos  => id,
                         wind     => A_window          );

  CREATE_WINDOW(X_window, row_size => 1, col_size => 1,
                          inmode   => W,
                          row_inc  => 0, col_inc  => 1,
                          ar       => X,
                          edge     => FALSE, edge_element => 0.0);
  SET(X_window, new_row => id, new_col => 1);

 solve_cycle:
  loop
   accept NEXT(x : VECTOR) do
     for i in 1..id
      loop
        partial_x(i) := x(i);
      end loop;
   end NEXT;

   X_window.win(1,1) := partial_x(id) / GET(A_sw,id,1);
   for i in 1..id-1
    loop
     partial_x(i) := (partial_x(i)
                      -GET(A_sw,i,1) * X_window.win(1,1));
   end loop;

   if id /= 1 then
       solve(id-1).NEXT(partial_x);
   end if;

   MOVE(X_window);

   exit solve_cycle when (EOS(X_window) = OUTSIDE);

  end loop solve_cycle;
 end BACK_SOLVE;
```

```
          begin    -- declare

             for i in 1..n-1
              loop
                solve(i).WHO_AM_I(i);
             end loop;

          end; -- declare

     CLOSE_LARGE_ARRAY(X);
     CLOSE_LARGE_ARRAY(B);
     CLOSE_LARGE_ARRAY(A);

  end MAIN;
```

1. Report No. NASA CR-172252	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle A Model for the Distributed Storage and Processing of Large Arrays		5. Report Date October 1983
		6. Performing Organization Code
7. Author(s) Piyush Mehrotra and Terrence W. Pratt		8. Performing Organization Report No. 83-59
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665		10. Work Unit No.
		11. Contract or Grant No. NAS1-17070, NAS1-17130
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546		13. Type of Report and Period Covered Contractor report
		14. Sponsoring Agency Code

15. Supplementary Notes Additional support: National Science Foundation Grant MCS78-00763.
Langley Technical Monitor: Robert H. Tolson
Final Report

16. Abstract

A conceptual model for parallel computations on large arrays is developed in this paper. The model provides a set of language concepts appropriate for processing arrays which are generally too large to fit in the primary memories of a multiprocessor system. The semantic model is used to represent arrays on a concurrent architecture in such a way that the performance realities inherent in the distributed storage and processing can be adequately represented. An implementation of the large array concept as an Ada package is also described.

17. Key Words (Suggested by Author(s)) large arrays concurrent programming distributed storage		18. Distribution Statement 61 Computer Programming and Software Unclassified-Unlimited	
19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 51	22. Price A04