REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Allocation Strategies for APL on the CHiP Computer		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) James L. Schaad		8. CONTRACT OR GRANT NUMBER(s) N00014-86-K-0264
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Washington Department of Computer Science Seattle, Washington 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Information Systems Program Arlington, VA 22217		12. REPORT DATE March 1987
		13. NUMBER OF PAGES 74
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

APL, CHiP, array allocation strategies

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This master's thesis describes a series of experiments in which APL programs were mapped onto the processor elements of nonshared memory parallel computers and their concurrent execution was simulated. The effects of selecting one of several different array allocation shcemes, different data routing policies and several processor topologies are reported. The general question of executing "shared memory programming languages" on nonshared memory parallel computers is briefly considered.

Allocation Strategies for APL on the CHiP Computer

James L. Schaad
Department of Computer Science, FR-35
University of Washington
Seattle, Washington 98195

TR 87-03-06

Abstract

This master's thesis describes a series of experiments in which APL programs were mapped onto the processor elements of nonshared memory parallel computers and their concurrent execution was simulated. The effects of selecting one of several different array allocation schemes, different data routing policies and several processor topologies are reported. The general question of executing "shared memory programming languages" on nonshared memory parallel computers is briefly considered.

The work described herein is part of the Blue CHiP Project and was supported in part by the Office of Naval Research Contract No. N00014-86-K-0264 and the National Science Foundation Grant No. DCR-8416878.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Programmers are accustomed to programming languages supporting shared memory; that is, languages that provide global variables and global data structures. However many recently developed or proposed parallel computers do not have a direct hardware implementation of shared memory. The lack of hardware support means that the support for shared memory must be provided in software. The problem is trivially solvable — run the program on only one processing element — unless the goal of high performance is also added. In this case, the problem reduces to allocating data to the separate processors and scheduling the data motion so that shared memory programs can be efficiently executed. Again there is a trivial solution — require that the programmer do the allocation and scheduling — unless the compiler is forced to do the allocation and scheduling. With the latter condition, we are confronted with a fairly challenging problem that has not yet been solved.

In this paper we focus on APL[12] as an existing, shared memory programming language to be run on a nonshared memory machine. APL, like FORTRAN, has a large user community, and thus there is some prospect that the results may be useful. Unlike FORTRAN and most other extant shared memory programming languages, APL has complex array operators that contain considerable inherent parallelism. Thus opportunities for concurrent execution can be derived from the semantics of the language, and therefore are directly applicable to *all* programs using these operators. This is in contrast to the complex dependency analysis of each individual program that must be done by systems like Parafrase[11] and PFC[2] in order to support FORTRAN.

We present a comparative analysis of two methods of mapping the APL semantics onto the CHiP (Configurable Highly Parallel) architecture[15]. Specifically, we discuss four different strategies for allocating arrays, the principle objects of APL, to the CHiP computer's processing elements. We then analyze the cost, in terms of execution time, of a representative sample of the APL array operators relating to the four schemes. Different allocation-operator pairs usually favor different interconnection structures and communication schedules, and hence yield different performance. Although this activity exhibits the value of the CHiP machine's configurability, it has several more fundamental benefits: First, because the CHiP computer is configurable and thus does not presuppose any particular communication structure, the semantics of the language dictate the proper structures. This means that the appropriateness of other host architectures can be easily deduced. Second, because the invention of an effective new programming language for non-shared memory parallel computers is such a difficult problem, the semantics of the inherently parallel APL operators can serve to illustrate constructs that may be incorporated into a new language. Third, the APL operators could be used as an intermediate representation for a dependency analysis for another language.

The work that has been done to date in implementations of APL have been for two different types of machines. A large body of work exists on how to implement

APL for sequential machines[5,8]. Abrams first proposed using a specialized architecture for evaluating of APL in 1970[1]. Since that time several implementations of APL have been done for vector processors, most notably by Budd[6].

This thesis will be divided into five chapters. In the first chapter an overview of this thesis and introductory comments on APL and the CHiP architecture will be presented. The second chapter will describe some of the problems with allocating variables on nonshared memory architectures and some solutions to these problems. Also the different allocation strategies considered for the APL system will be presented and the methodology for selecting an allocation strategy will be covered. The third chapter will present a proposed implementation of an APL system for the CHiP architecture. The fourth chapter will describe some simulations run in an attempt to start evaluating the effectiveness of some of the design decisions for the proposed APL system. The last chapter will present the conclusions, and make suggestions for further work.

1.1 APL

APL was created by Kenneth E. Iverson[10] while at Harvard as a language for mathematics. It was subsequently implemented for System 360 Computers at IBM[12]. APL was designed as a language with very powerful operators; often times one operator can take the place of several lines of code in a language such as FORTRAN. Thus the addition of two matrices may be represented in APL as $A + B$ and matrix multiplication is represented as $A + . \times B$. It is this power of the operators that makes APL ideal for this project: Many simple operations are contained within a single expression in the language.[1] Expressions in APL are evaluated from right-to-left as opposed to the normal left-to-right order. In addition there is no operator precedence; all precedence is obtained from explicit parenthesization.

As seen above, APL operators work on the entire contents of a variable, independent of either the number of or type of values referred to by the variable. In this light the subscripting of an array can be considered as applying a function to a variable.[2] The number of dimensions and the values of the dimensions are dynamically changeable at runtime. A scalar is simply an array of 0 dimensions containing a single value, so the scalar value 1 has a different structure than the vector (1-dimensional array) with the single element 1 even though they appear to be the same for certain operations such as displaying.

The rest of this section will describe three common APL operators: dyadic operators, shape and reduction. This discussion is by no means complete, a more through introduction to APL may be found in other sources[7,12].

The single most commonly used operation in APL is the **dyadic element-by-element** "operator". This is actually a group of several operators including matrix

[1]An additional benefit of representing multiple operations as a single expression is that the operations are much easier to recognize. Consider the two code segments below that compute the expression $D = A(BC)$ where A, B and C are all NxN arrays.

```
        APL                       FORTRAN
                              DO 10 I = 1,N
                              DO 10 J = 1,N
                              D(I,J) = 0
                              DO 20 K = 1,N
  D ← A+.×B+.×C               T = 0
                              DO 30 L = 1,N
                           30 T = T + B(K,L) * C(L,J)
                           20 D(I,J) = D(I,J) + A(I,K) * T
                           10 CONTINUE
```

[2]Abrams used this approach in designing his APL machine[1].

addition. The group is made up of all operators which perform an elementwise combination of values in the same index positions. While the exact meaning of the expression $A + B$ depends on both the type of, and the dimensions of the variables A and B, the same basic algorithm is followed in evaluating all possible combinations. If the shapes (dimensions) are equal then the arrays are said to conform and the elements of the arrays are added together. If one of the variables is a scalar then its value is added to all elements of the other variable. If neither of these conditions is met then the expression is said to be non-conforming and an error condition is raised. Examples of some element-by-element operators may be found in Figure 1.1c.

The **shape** operator, represented by $\rho$is a monadic operator which takes an array and returns the dimensions of the array in a vector. One common use of the shape operator is to get the number of dimensions of an array. To do this the function is applied twice, *i.e.* $\rho\rho A$, and returns a scalar. One interesting result is the application of shape to a scalar, the result will be a vector of length 0. Applying shape again will result in the scalar number 0. Examples of the shape operator are in Figure 1.1b.

The **reduction** operator / takes as its arguments a dyadic scalar function $\mathcal{F}$ and an array A; it places the function $\mathcal{F}$ between all elements of the array along the last dimension and evaluates the resulting expression. The application of the reduction operator results in an array with one fewer dimensions than the source array. Examples of the reduction operator appear in Figure 1.1d.

$$A = 5 \qquad B = (6\ 1\ 3\ 5\ 9) \qquad C = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 3 & 5 & 7 \\ 1 & 5 & 9 & 13 \end{pmatrix}$$

(a)

$$\rho A = (\ ) \quad \rho\rho A = 0 \qquad \rho B = (5) \quad \rho\rho B = 1 \qquad \rho C = (3\ 4) \quad \rho\rho C = 2$$

(b)

$$C + C = \begin{pmatrix} 2 & 4 & 6 & 8 \\ 2 & 6 & 10 & 14 \\ 2 & 10 & 18 & 26 \end{pmatrix}$$

$$C \times C = \begin{pmatrix} 1 & 4 & 9 & 16 \\ 1 & 9 & 25 & 49 \\ 1 & 25 & 81 & 169 \end{pmatrix}$$

$$A + C = \begin{pmatrix} 6 & 7 & 8 & 9 \\ 6 & 8 & 10 & 12 \\ 6 & 10 & 14 & 18 \end{pmatrix}$$

$$A \times C = \begin{pmatrix} 5 & 10 & 15 & 20 \\ 5 & 15 & 25 & 35 \\ 5 & 25 & 45 & 65 \end{pmatrix}$$

(c)

$$+/B = (6 + (1 + (3 + (5 + 9)))) = 24$$

$$+/C = (10\ 16\ 28)$$

$$\times/B = 810$$

$$\times/C = (24\ 105\ 585)$$

$$+/[1]C = (3\ 10\ 17\ 24)$$

$$\div/C = (0.375\ 0.238\ 0.138)$$

(d)

Figure 1.1: **Examples of APL operators** (a) The arrays to be used in the examples. (b) Examples of the shape operator. (c) Examples of dyadic element-by-element operators. (d) Examples of the reduction operator.

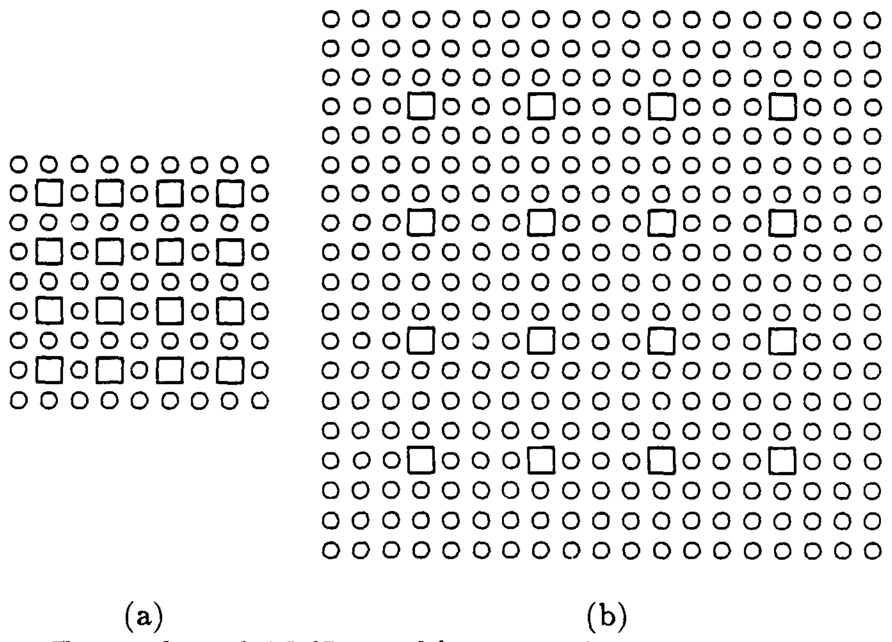
Figure 1.2: **Examples of CHiP Architectures** Two examples of CHiP architectures are presented above. The number of processors on a side and the number of switches between processors are two parameters describing the computer. (a) A CHiP computer with 4 processors per side and a corridor width of one. (b) A CHiP computer with 4 processors per side and a corridor width of three.

1.2 The CHiP Computer

The CHiP computer[15] was proposed by Larry Snyder while at Purdue University. It is a nonshared memory, MIMD parallel computer with a configurable communication structure. The system can rapidly switch between any of several user defined interconnection structures under program control.

The internal structure of the CHiP architecture is formed by arranging the processing elements (PEs) on the points of a grid to form a processor array. Switch elements are placed between the processors in columns and rows as shown in Figure 1.2.

The number of switches between processors is called the corridor width and

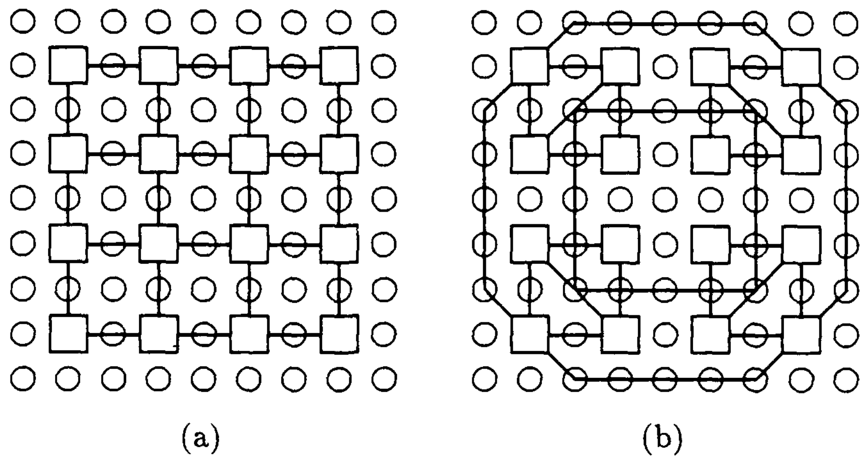

Figure 1.3: **Examples of Interconnection Structures on the CHiP Architecture:** Two different interconnection structures on the same CHiP architecture are shown here; (a) shows a simple two dimensional grid while (b) shows a binary 4-cube.

may be assumed for the purposes of this discussion to be large enough to hold any communication structure. When programmed, the switches provide point-to-point communication paths between processors.[3] Examples of programmed paths are shown in Figure 1.3.

For the purposes of this paper the skeleton or control structure of the CHiP computer is just as important as the internal communication paths. A single external processor is used to control the execution of the array by loading information into the processor array either by a broadcast or by sending information to a single processor. It is also used to load the interconnection structure(s) into the switches and to control the order of phase execution.

The applications to be solved on parallel computers are usually very complex. One way in which they can be made simpler is to decompose them into a series of

[3]This is as opposed to the packet switching of data as is done in the Cosmic Cube[14].

smaller (and hopefully easier) sub-problems. In the CHiP philosophy, a phase is the term used to describe the smallest unit of parallel computation. A phase will then correspond to one sub-problem, while a series of phases (possibly repeated) will then correspond to the entire problem. Different interconnection structures will often provide both different algorithms and running times to solve problems. An interconnection structure that is optimal for one sub-problem may not be for a second sub-problem. The concepts of phases and reconfigurability allow for the use of multiple interconnection structures (one per phase or sub-problem). For example, a simple relaxation problem may be decomposed into three different sub-problems or phases.

1. The first phase is the point relaxation; the communication pattern is to exchange information with all nearest neighbors and therefore would use a grid structure.

2. The second phase computes a global relaxation value; this would be done using a tree structure.

3. The last phase distributes the global value to all processors; this may also be done on the same tree structure.

Thus a simple problem is solved more efficiently by using two different communication patterns than by using just one. The algorithms presented in this thesis are composed of multiple steps, each having its own, possibly unique, communication structure.

Chapter 2
Allocation Problems

Allocation and accessing of data in memory is a mechanical process for sequential machines. For arrays of data, this is done by mapping the elements of an array into successive locations of memory and constructing a function that can compute the physical address of the data element in memory given the dimensions of the array and the indices for an element to be accessed.

The first parallel machines that were introduced were vector machines and MIMD machines with shared memory. In both of these cases the problem of initially allocating the data became slightly more complicated. For example, in a vector machine best results are obtained if adjacent elements are not in the same bank of memory, but the access function could remain the same since all processors could address all data elements. The differences between the simple sequential machine and these machines could generally be hidden by the hardware.

With the advent of MIMD nonshared memory computers the problem of data allocation has become much more complicated than it was for the sequential computer. A new approach to allocation and accessing both scalars and array variables must be designed for MIMD nonshared memory architectures such as the CHiP computer.

This chapter will deal first with the allocation problem for a general shared memory programming language and then with the allocation problems for the specific case of implementing the APL language.

2.1 Allocations for MIMD Nonshared Computers

Due to the lack of a single monolithic address space, no address in memory is accessible from all processors. In fact, each address in memory is generally addressable from only a single processor. This partitioning of memory requires that the access algorithm be rewritten to include protocols for reading from and writing to memory locations attached to other processors.

A new read algorithm allowing for the protocols necessary for accessing data elements from other processors may go something like the following.

1. Determine if data element D is contained in the local memory. If so then the data is read and the value is returned.

2. Otherwise, broadcast a request to all processors in the processor array to obtain the correct value of the data element D.

3. Receive zero, one or more responses. If no responses are received then repeat step 2. If more than one response is received then select the correct value based on timestamp information.

This algorithm is incomplete in two ways. It is possible it may never terminate if no responses are ever received in step three. Also the algorithm does not know how to determine when more than one value should be expected. Additionally each processor must expend some of its running time to service read and write requests from other processors; moreover either the processor or the switch must do forwarding of request and data packets.

The read algorithm as presented above will work in the general case. The broadcast is necessary to ensure finding the data although it is extremely inefficient. The efficiency of the read algorithm can be improved substantially. There are several places in the algorithm where this may be done. These modifications fall into two major categories. First, the number of request messages sent in step two may be decreased by predicting the location of the data. Second, the distance the data travels may be decreased initially allocating the data near the processors which use it. Third, if the predictive function allows data to be moved, how does it keep the data close to the processors which will use it. This will not be covered in this thesis.

2.1.1 Predictive Functions

The use of a predictive function in step two increases the efficiency of the algorithm by avoiding a full broadcast. Instead the requesting processor sends a single message or a small broadcast just to those processors in which the data is expected to reside. This will reduce both the number of read requests that are in the system and the number of local memory searches which will fail due to the element not being present.

Two classes of predictive functions are available: static functions and dynamic functions. Static predictive functions are pre-programmed with the locations of each data value. Static predictive functions must therefore be tailored to both the specific program, and the specific system on which it is run. Dynamic predictive functions are able to learn the locations of the data values while the program is running, and adapt to new access patterns while the program is running. Dynamic access functions have three components:

1. A means to construct local tables to improve the "best guess" of which processor(s) contains a data element.

2. A means to allow data elements to be moved between processors and a method of either forwarding requests to the new processor or informing other processors that the data has been moved.

3. A means to allow for either replication or cacheing of frequently used data values in local memories. This component is often not implemented as it can consume more resources than it frees.

2.1.2 Initial Allocation of Variables to Processors

The second class of improvement deals with the initial allocation of data to processors. The closer a data value is to the processors that use it, the less time the data value will spend in transit. This is reduced to zero time if the data value is actually in the processor's memory.

Replicating a variable in every processor will provide the most drastic speed-up for the read algorithm. In this case the local search will always succeed in finding the data value and no read requests will need to be made to other processors. However this simplification of the read algorithm causes the write algorithm to become correspondingly more complicated. The write algorithm must now update the value in every processor rather than in a single processor. Still, this is an extremely useful strategy for those variables which are read frequently by every processor and updated only infrequently.

If variables (or array elements) are generally used only by a single processor then much of the same improvement as is provided by replication may be made by placing that variable in the local memory of that processor; without the added complexity of updating a replicated variable. This can be a difficult problem since it involves predicting which processors will access the data. A simple predictive function leads to poor execution times due either to a large number of messages in the system or to a mismatch between the usage of variables and their locations. Thus the predictive function must take into account the usage patterns of the program, which are generally difficult to obtain except in the simplest cases.

Grid Relaxation Problem

One type of program where the system actually could do the data flow analysis is for an n-dimensional grid relaxation problem. In these programs there are a few scalar values which are used globally and many arrays of data (or scalars which are used locally only). The global scalars can be replicated in every processor since the rate of updates is generally low when compared to the rate of reads, and the arrays of data can be spread out over the processors.

Each data point in the problem interacts only with the data points immediately adjacent to it in each dimension, so the processors only need to obtain data points from the immediately adjacent processors. The interconnection structure used with these programs reflects this data flow pattern, that of an n-dimensional grid. The program can therefore allocate the array data onto the interconnection structure in a logical manner, that is, map each point in the problem space onto a point in the interconnection structure such that adjacent points in the problem space remain adjacent.

A programmer could also make a second optimization to the program which the system would not see. Since every processor in the array is executing approximately the same code, every processor can predict exactly when each piece of data in its local memory will be needed by adjacent processors and anticipate their needs and supply the data value without the need for a request. This means that steps one and two of the read algorithm as presented could be eliminated.

2.1.3 Avoiding Program Analysis

In the case of an arbitrary program it is not possible to do a mechanical analysis of the usage patterns of the program in reasonable time. At this point there are two options to consider: First, the programmer may be required to supply either hints or complete information about how the data assignment should be done. Second, the language can be re-implemented to provide operator primitives which may be analyzed for data flow independent of any actual program.

Maximum Flow Problem

The maximum flow problem in a directed graph illustrates how the second approach may work, that is, to provide a program or primitive which may be analyzed independent of the data. The maximum flow problem takes as input a labeled directed graph. The label represents a measure of flow along the edge. The maximum possible flow from the sources to the sinks is then computed.

One parallel solution of the maximum flow problem would be to map the nodes of the graph onto the processors, one node per processor. The edges of the graph would then be embedded in the interconnection structure of the parallel computer.

On the CHiP computer this could be done by embedding the problem graph in the switch structure. Each node of the graph would then total the incoming flow on its edges and send out messages on the input edges and output edges either limiting the input flow or the specifying output flow. This would continue until all processors agree that a solution has been reached.

The problems with this solution are: First, each problem graph needs to be embedded into the communication graph provided by the architecture. An analysis of the graph would be needed to be done in the embedding which would be comparable to the data flow analysis done normally. Second, the termination condition is not well defined.

A second approach to the maximum flow problem is similar in that the algorithm still sends messages up and down the graph to compute the maximum flow of the graph. However it differs in two distinct ways. First, the solution is now a general solution, the code and interconnection structure are independent of the problem graph. Second, there is now global control of the flow of information on the edges of the problem graph. All data motion is cycled first down from the sources to the sinks and then back up from the sinks to the sources. This allows a first approximation of the data flow of the program to be made independent of the actual problem graph. For a dynamic prediction function this allows one to make a "good enough" first guess in allocating the data that it will generally not need to move far. This can be done on a global basis without analyzing a specific graph structure.

This concept of doing the data flow analysis independent of the data will be exploited even more with the APL operators. Each operator can be broken down into a set of cases depending on the input data and each case can be completely analyzed independent of the actual data.

2.2 Dataflow for APL Operators

As previously shown, in order to run an arbitrary program efficiently on an MIMD nonshared memory machine, predictions must be made in order to allocate the data to the processor memories. APL provides some interesting insights in how to avoid a general case analysis of data flow patterns on a program specific basis. Instead of doing an analysis for every program, data flow analysis is done for each APL operator. This can be done since the APL operators incorporate a much larger view of the program's activities than is provided either by a set of codes to be executed in different processors or even a single code to be executed in every processor. Operators such as drop, inner product, and outer product include, as part of their definitions, information describing how different elements of the arrays are to be combined and thus can be used to predict the expected data flow of the program *independent of either the specific data values used or the program containing the operator.* This allows the system implementor to do a complete analysis of all possible data flow patterns for all APL programs without running a program or even writing the program. Therefore one can predict *all* possible ways that the data will interact without having to look at the specific program under consideration.

It is expected that two different operators may have different optimal allocations in the processor array. When this occurs we must look at the expected frequency of use for the operators and the cost of using an operator in a non-optimal allocation scheme to choose a mapping algorithm that will be optimal when considering all operators.

2.2.1 APL Operators Classified by Data Flow Patterns

Below is a description of the data flow patterns found in APL and the operators which cause them.[1]

1. The first set of operators have no data motion associated with them; the values are simply produced from thin air and magically show up in the correct locations. These operators are shape and iota. For example, once the operator iota knows its parameter, each processor can independently produce those data elements which the mapping algorithm would have assigned to its local memory.

2. The monadic element-by-element operators have no data motion either as the operator uses a single data point, reading the parameter and writing the result to the local memory. An example of this is unary minus, each data element can be independently negated.

3. The dyadic element-by-element operators have no data motion associated with the evaluation of the operator. However the operator does require that both arrays involved in the operation be laid out using the same allocation scheme so that the pairs of values reside in the same processors. Examples of these operators are dyadic plus, minus, divide and times.

4. The fourth set of operators restrict the data motion to be within a single dimension of an array. These are the axis operator group. The data values participating in the operation lie along lines in the array parallel to a specified axis. Which axis is to be paralleled is a parameter to the function. Each

[1]We will omit consideration of all special cases of operators as many of these are uninteresting data flow patterns. Most of the special cases have no data motion associated with them.

axis may be specified, although it is usually the first or the last. These operators are: decode, encode, concatenate, reduction, scan, compression, and expansion.

5. The fifth set of operators require a possible matching of every data point in both parameters. The data motion of this group of operators would require a method for every pair of data points to be combined. This may be done independent of any specific mapping algorithm. These operators are: membership, index of, and outer product.

6. The sixth set of operators is used to modify the arrangement of or the indices of the elements. To bring the data back into the allocation scheme may require a complete permutation of the data. The first argument, usually a small APL array, is used to modify the index set of the second argument. These operators are: take, drop, rotate, reversal, reshape, transpose, ravel, laminate, and array indexing.

7. Inner Product is a special operator in its own class. It combines two different dimensions of two variables. It can be programmed in a similar style to the systolic matrix multiplication algorithm. The data motion in this case is in the last dimension of the first array and in the first dimension of the second array.

8. The last set of APL operators uses unknown data flow patterns. These operators are: format, matrix inverse, and matrix divide.

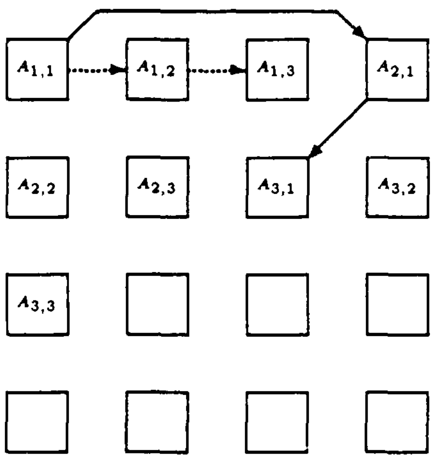

Figure 2.1: **Ravel Order Allocation** - This figure shows a 3x3 array allocated on a 4x4 processor array. The solid line shows the data flow for the element $A_{3,1}$ doing a scan in the second dimension. The dotted line shows the data flow for the element $A_{1,3}$ doing a scan in the first dimension.

2.3 Candidate Allocation Models

We consider three possible allocation schemes: ravel (row-major) order, n-dimensional grid and projected dimensions. An a modification applicable to all the allocation schemes. In selecting which allocation scheme(s) to use the data flow patterns of the APL operators presented above must be considered.

2.3.1 Ravel Allocation

The ravel allocation conceptually arranges all the processors in a line and places the data elements into the processors. This scheme works well for the reshape operator where the order of the elements is maintained, but is definitely hampered by such operators as scan or rotate which may be done in the second or higher dimension of the array. In these cases it requires that every processor be able to communicate with processors distance i away from it, where i is a function of the

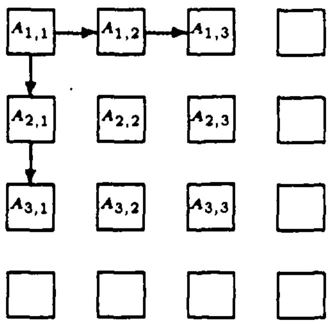

Figure 2.2: **N-Dimensional Grid Allocation** - This figure shows a 3x3 array allocated on a 4x4 processor array. The solid line shows the data flow for the element $A_{3,1}$ doing a scan in the second dimension. The dotted line shows the data flow for the element $A_{1,3}$ doing a scan in the first dimension.

array size and can not be determined until run time. For example, if a scan is done in the second dimension on the array in Figure 2.1 then the element $A_{1,1}$ must be sent to $A_{2,1}$ and then to $A_{3,1}$ following the solid line. If the scan were done in the first dimension then the data flow for element $A_{1,3}$ would follow the dotted line.

Communication in dimensions other than the first is especially important for the axis operator group, the default case for these operators is to use the last dimension of the array. For matrices the second dimension (the solid line) is the default. This leads to the need for message forwarding systems to support these operators in the ravel order allocation scheme since the correct distance to send a value can not be known until runtime.

2.3.2 N-Dimensional Grid

The ability of the APL operators to work in any dimension makes it desirable to have an allocation scheme which will not favor any one dimension. One such scheme would directly implement the structure of an APL variable as an n-dimensional array. This scheme allows the implementor to ignore the problem caused by evaluation of the same operator on different dimensions. The same algorithm can be used in every dimension. The only thing that will need to be altered is interconnection structure, one for each of the possible dimensions.

Using the same example as above, Figure 2.2 shows as a solid line the data flow needed to compute the element $A_{3,1}$ while the dotted line shows the data flow needed to compute the element $A_{1,3}$.

The n-dimensional allocation scheme does however have its drawbacks. The operator reshape which alters the length and number of dimensions without rearranging the order of the data elements can causes a large amount of reshuffling of the data values.

2.3.3 K-Dimensional Projection

The k-dimensional projection is almost dimension independent. It has two different types of dimensions that must be dealt with. An operation that occurs along a dimension may be contained entirely within a processor if that dimension has been projected out. Or dimensions may go across multiple processors as in the n-dimensional grid case. This scheme is equivalent to the n-dimensional case for array dimensions of k or smaller.

Figure 2.3 shows an example of a 3-dimensional array mapped using a 2-dimensional projected allocation scheme. This projection ignores the second dimensions of the array but could just have easily ignored either the first or the third dimension. Looking at how the data flow for a scan would work in this allocation scheme, the dotted line shows the data flow to compute the scan for the element $A_{1,1,3}$ (the first dimension), the solid line shows the data flow to compute the element $A_{3,1,1}$

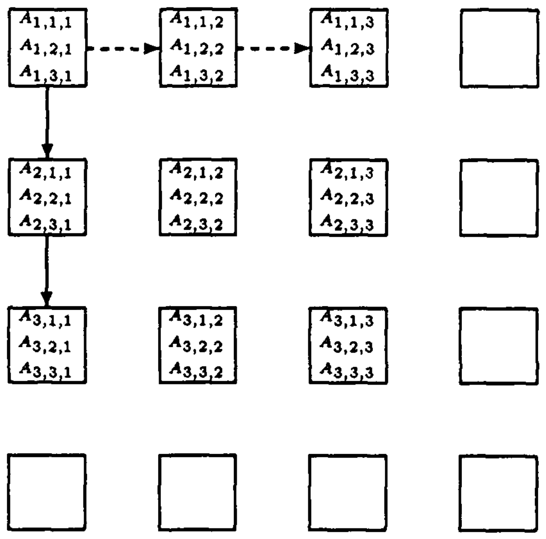

Figure 2.3: **K-Dimensional Projection Allocation** - This figure shows a 3x3x3 array allocated on a 4x4 processor array. The solid line shows the data flow for the element $A_{3,1,1}$ doing a scan in the third dimension. The dotted line shows the data flow for the element $A_{1,1,3}$ doing a scan in the first dimension. For the scan in the second dimension no data movement is needed since all the elements are already in the same processor.

(the third dimension). For each of these cases three values must be sent over each communication channel used. No data flow would need to occur for evaluating a scan in the second dimension (element $A_{1,3,1}$) since all of the data values which are needed are already present in the same processor.

This allocation scheme has many of the same characteristics as the n-dimensional grid scheme. As previously noted they are actually identical for any array with dimension less than or equal to k. It should perform better for reshape if the dimension is greater than k since much of the data motion will be contained with-in a single processor. This allocation scheme however has the disadvantage of not being completely uniform. Two different methods of evaluating many of the operators are required due to the two different mappings of a single dimension (in a processor and across processors).

2.3.4 Stay-Put Variation

Allocation schemes such as the n-dimensional grid have a large amount of interprocessor communication when evaluating such operators as reshape. When a series of these operators is strung together an element may pass through several processors (and the path may either return to or pass through the original processor). The APL operators in the sixth group (section 2.2.1) modify the indices of the array. This can be done independent of the actual location of the element. If a data element were to consist of the index set of the element as well its value, these operators could modify the index set without changing which processor contained the data element. Operators such as the dyadic element-by-element operators which require that two arrays use the identical allocation must then move the elements of the arrays to match the allocation scheme. This is done based on the indices in the

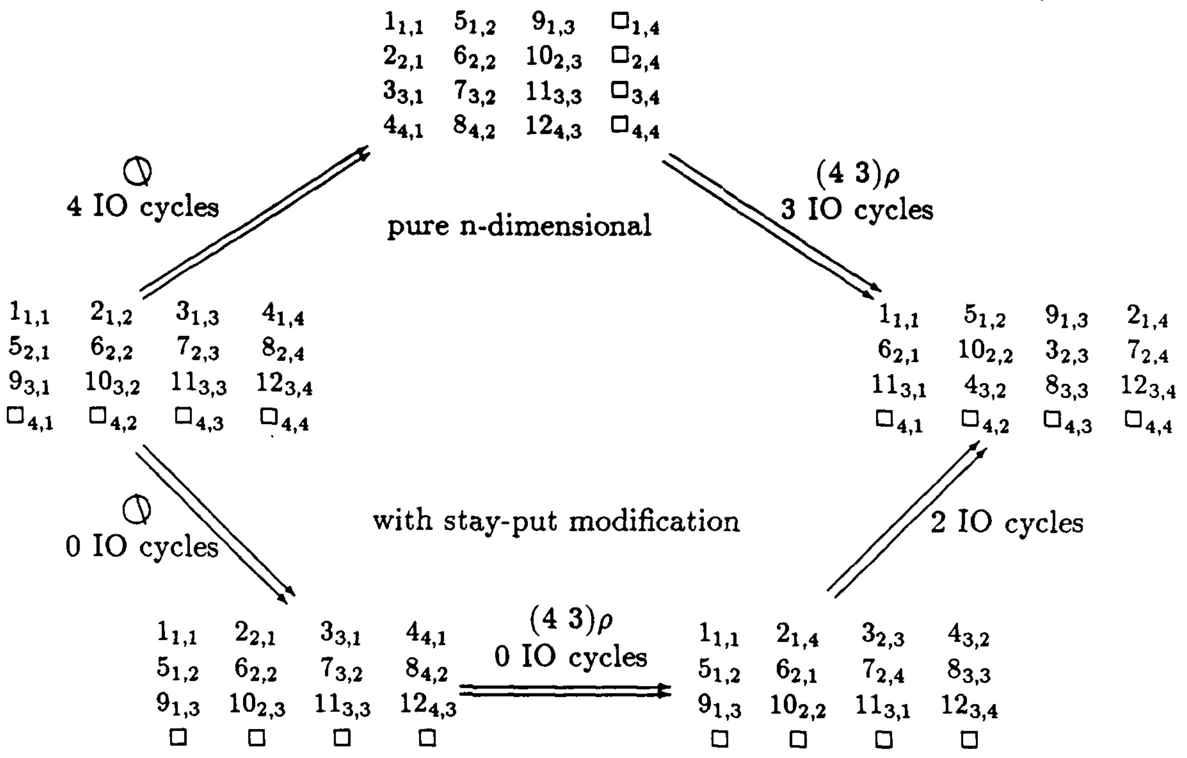

Figure 2.4: **Example of Stay-Put Modification** – This figure shows the data movement of the elements of an array both omitting the stay-put modification (the upper arc) and using the stay-put modification (the lower arc). In both cases the underlying allocation scheme is the n-dimensional grid. A transpose (⍉) followed by a reshape ((4 3)ρ) is evaluated. The top arc uses seven IO cycles in computing the result. The bottom arc, with the stay put modification, takes one more step in evaluating the expression. However only two IO cycles are needed to evaluate the same expression. In both examples all communication is done on a binary n-cube.

data elements.

The Stay-Put modification to an allocation scheme is also applicable to APL operators not in group six. The operators in group two are applied to a single array, and therefore can be done independent of the allocation scheme. The APL operators in group five match up all elements from both arrays. This may be programmed to be done independently of any allocation scheme.

An example of how the stay-put modification can reduce the amount of interprocessor communication can be found in Figure 2.4. In this example the same operators are applied to an array using the n-dimensional allocation scheme. The lower arc uses the stay-put modification to reduce the number of IO cycles from seven to two. This is the type of savings we would hope for but it can not be guaranteed as it is dependent on both the data and the program.

This modification to an allocation scheme represents a potential savings in communication costs. Three factors will decrease the potential savings: First, since a data element now consists of a set of indices as well as the value, each communication is more costly as a larger number of bits must be transferred between processors. Second, the savings in communication costs assumes that the operators which may be done independent of the allocation scheme will occur together in APL programs. Third, the cost takes into account only the number of communications, if all the data elements are in a single processor the number of communications is reduced to zero but with a loss of all parallelism during execution of the program. It should be noted that the last two restrictions may be in conflict, that is the longer the sequence of operators, the more likely that fewer processors will be involved in the evaluation (or that a processor will have a correspondingly larger percentage of the data elements).

2.3.5 Decision Criteria

In order to make a selection between the different allocation schemes, a set of decision criterion must be established. The objective is to choose the allocation structure which will have the fewest overall IO cycles, while at the same time providing the maximum amount of parallelism, during the evaluation of an APL program. For any given program it seems possible that any of the above allocation schemes may be best. For example a program consisting of only reshape operators will favor the ravel-order allocation scheme, but a program consisting only of scan operators will favor the n-dimensional grid allocation.

The analysis of the APL operators by data flow patterns done in Section 2.2.1 is useful in deciding upon the allocation scheme. A data flow pattern will generally favor one of the allocation schemes, or be independent of all of the allocation schemes.

The operators from groups one, two and three require no data movement to be made. These operators will therefore not favor or disfavor any of the allocation schemes. The only requirement put on the allocation scheme is by group three. These operators need the allocation for both variables to be consistent.

The data flow for the group four operators is along one axis of the array. If the first axis is used then no preference is seen. However for the higher axes a preference for the n-dimensional grid or the k-dimensional projection is found.

The APL operators in group five do not exhibit a preference for any of the allocation schemes. The need to match up all elements from both arrays will take an equivalent amount of time independent of the allocation scheme.

The operators in group six will perform best if the stay-put modification is used. If the stay-put modification is used, then no preference for the underlying allocation scheme will be found. It will be most effective if multiple operators may be applied without reorganizing the data array to the underlying allocation scheme. If the stay-put modification is not used then two different sets of preference are found. The operators take, drop, rotate, and reversal will prefer the n-dimensional grid or the k-dimensional projection allocation schemes. The operators ravel and reshape will prefer the ravel order allocation scheme to be used. Array indexing by vectors

Table 2.1: **APL Usage Patterns** – Table taken from [13]

Operation	Percentage of use
Scalar primitives	73
Subscripting	18
User function calls	3.6
Reduction	2.6
Mixed output	1.0
Explicit axis specification	0.6
Inner product	0.4
Outer product	0.4
Scan	0.1

will prefer the n-dimensional grid or the k-dimensional projection. Indexing by either scalars or arrays does not show a preference for any allocation scheme; this is a problem no matter which scheme is used.

The inner product operators can be divided into three cases. If one operand is a scalar or both are vectors then the data flow pattern for the inner product is equivalent to that for a reduction operator. If both operators are 2-dimensional arrays or smaller then the n-dimensional grid or k-dimensional projection will be preferred as the systolic matrix multiply can be directly implemented. If either argument is greater than a 2-dimensional array then it will need to be reshaped into a 2-dimensional array before the operation can proceed. This will not favor any allocation scheme.

The last group of APL operators, those in group eight, have no known preference for an allocation scheme. In part the reason is because the data flow patterns for these operators is unknown.

In order to make any assumptions about how a general APL program will execute the overall usage of the different APL operators must be known. Several studies have looked at the occurrence of APL operators both statically[13,3,9] and dynamically[4]. The results of one such study are presented in Table 2.1. At a first glance, this seems to be a rather discouraging picture for choosing an allocation scheme. Scalar primitives can be computed in the same amount of time (zero IO cycles) for all allocation schemes and subscripting is difficult to do in any of the allocation schemes.

The overall preference of the operators is for either the n-dimensional grid or the k-dimensional projection allocation schemes. As an implementor a preference is found for the n-dimensional allocation scheme. Only a single algorithm for evaluating each operator needs to be designed and implemented. Among other things, this will lead to a smaller set of support code in the processor arrays.

2.4 The Proposed Allocation Model

2.4.1 Description of Model

The proposed allocation scheme can be broken into two portions. The first is the basic underlying allocation scheme and the second is a generalization of the scheme to decrease the number of communications needed. The underlying allocation scheme is based on an n-dimensional grid mapped onto the processors that reflects the structure of an array directly in the allocation scheme. The n-dimensional grid was chosen since it neither favored nor disfavored any dimension. Since the allocation scheme was developed with the CHiP computer in mind, it should be noted that the use of the n-dimensional grid allocation scheme does not enforce the use of an n-dimensional interconnection structure. The use of other interconnection structures that may have better evaluation algorithms, such as a tree structure for reduction, may be used freely as long as the basic data point to processor assignment for the

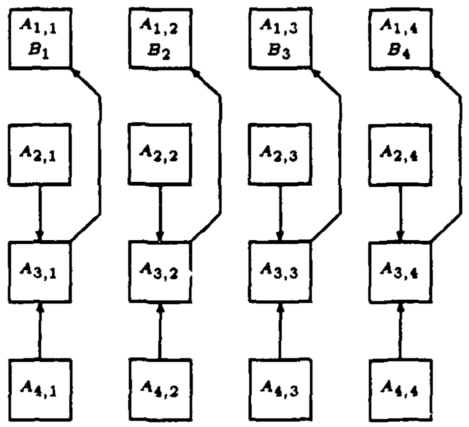

Figure 2.5: **Mismatch of Allocation Scheme and Interconnection Structure:** This figure shows how a reduction can be computed along the second dimension of the array A. The array A is laid out in the array using an n-dimensional allocation scheme. The reduction in the second dimension uses a set of tree interconnection structures to compute the resulting array B.

n-dimensional grid is observed. An example of how this works can be found in Figure 2.5. This figure shows how a reduction in the second dimension could be done using a set of tree interconnection structures on an n-dimensional allocation scheme. The result will be a vector across the processors in the top of the processor array, each element in the result is computed by adding the four values from the column below it using the tree to combine the values in *log* time.

The proposed allocation scheme will also use the stay-put modification to decrease the number of communications needed. This relaxes the normally strict allocation scheme so that data values may be in processors other than those which the allocation scheme would require. This allows for functions such as transpose or reshape (which would be implemented as modifications to the access function for the array[6] in a serial machine) to be evaluated by modifying the coordinates of the data value rather than actually moving the data value between processors. The strict underlying data allocation model will then be enforced only when evaluating those operations such as the dyadic element-by-element operators which require it.

2.4.2 Restrictions of the Model

In attempting to implement our ideal allocation scheme on real hardware two restrictions must be imposed. Both of these restrictions arise from the mapping of the model, which allows for an infinite number of processors, onto the real hardware, which has only a finite (and fixed) number of processors. The two different methods of getting an infinite number of processors in the model are: First, there may be an infinite number of dimensions in the data array. Second, there may be an infinite number of data elements along one or more dimensions. A modification must be made to the model to deal with each of these restrictions.

The language definition of APL actually restricts the number of dimensions of an array to 63 rather than the infinite number proposed in the model. Even this limit is too large for today's hardware; an array with just two elements in each dimension would require 2^{63} (or about 10^{19}) processors. To deal with this problem we modify the allocation scheme to use the k-dimensional projection rather than the n-dimensional grid. When the number of dimensions of an array exceeds a set limit k, then the excess dimensions of the array will be projected onto a single processor. Experimentation will be needed to determine the value of this limit and which dimensions should be projected.[2]

A second restriction on the model is that it assumes that there are sufficient processors available for each data value to be mapped to a single processor. This

[2]It is expected that the value of k should be between 4 and 8. A value of 4 would allow for an outer product to be executed with two 2-dimensional arrays without having to project the result. The value of 8 the maximum value which can provide a matrix of reasonable size under today's hardware restrictions, and is probably larger than is required for all but a few APL programs.

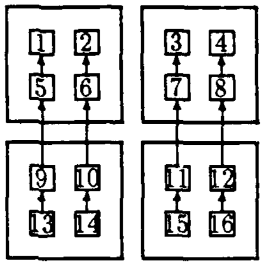

Figure 2.6: **Sub-array Coalescing** – This figure shows a sub-array coalescing of a 2-dimensional array. In this example two communication channels between processors must be multiplexed onto a single hardware channel.

would require an infinite number of processors in each dimension of the grid. In reality there are a fixed number of processors in each dimension and the allocation scheme must deal with arrays that are larger than this size. The model is modified to allow for *the assignment of multiple data elements to a single processor.* Each dimension in the array is treated independently for the purposes of coalescing data values. Two different modifications may be made to the model: First, adjacent values along the dimension may be placed into a single processor. This is the sub-array coalescing method. Second, if there are k processors in the current dimension, then every k-th data element may be placed in the same processor. This is the modulo coalescing method.

An example of how the sub-array coalescing method works can be seen in Figure 2.6. One advantage of this method can be seen in the figure, the total number of interprocessor connections has been decreased. Some of the channels in the original model are now contained in a single processor. The disadvantage of this method of coalescing is that the process of determining which processor a data element is mapped to requires knowing the indices of the data element, the dimensions of the

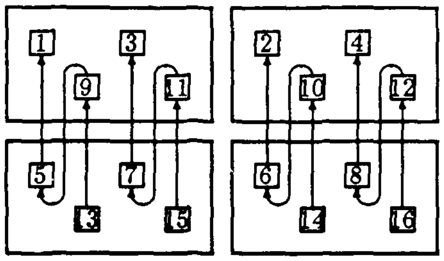

Figure 2.7: **Modulo Coalescing** – This figure shows a modulo coalescing scheme of a 2-dimensional array. In this example six communication channels need to be multiplexed onto a single hardware channel.

hardware and the dimensions of the array. The dimensions of the hardware may be hardcoded into the processors and the data element indices are associated with the data element but the dimensions of the array must be broadcast from the controller processor.

An example of the modulo coalescing method may be found in Figure 2.7. The advantage of this method is that the mapping function requires only the indices of the element and the dimensions of the hardware, both known values. The broadcast of the dimensions of the array can be avoided. The disadvantage of this method is that all of the interprocessor connections present in the original model are still present. However since there are now multiple data elements per processor, there are also multiple communication channels for every hardware communication channel.

The use of the sub-array coalescing method in the first dimension does not imply that it need also be used in the rest of the dimensions. The two methods of coalescing may be intermixed freely. One possible way to take advantage of this would be to use the sub-array coalescing in the first and last dimensions of the array and the modulo coalescing method in the rest of the dimensions. This allows the system to use the decrease in the number of communication channels for the two dimensions in which most of the operators occur. At the same time the amount of information needed to be broadcast to compute the new mapping of an array would be minimized to two values.

2.4.3 Other Questions

There are other unanswered questions about the model which will require experimentation: (1) How evenly distributed are the elements of an array over the processor array if one uses the stay-put modification? (2) In mapping an array variable onto the processor array should all the dimensions of the hardware be the same length? Should the dimensions even be fixed values or should they vary as the size of the array varies? (3) Are there other more efficient algorithms for evaluating the APL operators and how would they affect the selection criteria presented above.

Load balancing or the evenness of the distribution of the array elements in the processor array is extremely important. The more even the distribution, the higher the degree of parallelism that is realized in the system. A processor with data does useful work while a processor without data is idle. The introduction of the stay-put modification to an allocation scheme decreases the likelihood that an array will be evenly distributed over the array.

Since the stay-put modification to an allocation scheme does not force data elements to move between processors, as time passes a large number of data elements may build up in a single processor. The build up will be decreased by frequent execution of the operators which do a strict enforcement of the underlying allocation scheme. If the data balance becomes a problem, then how is it to be remedied? One possibility is to make some of the operators (such as assignment) enforce the underlying allocation scheme when they are executed. A second possibility is to introduce a new operators into the code which does nothing but to enforce the underlying allocation scheme.

The dimensions of the logical hardware may change with the dimensions of a variable. This can lead to an improved load balance and therefore to a faster system. An example of how the logical hardware dimensions affect the load balance can be seen in 2-dimensional simulation system. The major data array will consist of perhaps several thousand pairs of data (x and y values). On an 8x8 hardware array this leads to using only 25% of the available processors. If, instead of an 8x8 logical hardware array, a 32x2 logical hardware array is used all processors will be used in computing the results.

A second interesting possibility is to use non-uniform lengths for the hardware array. The first and last dimensions may be increased at the expense of the middle dimensions. Since many operators occur along these axes it maybe that the overall execution time may decrease.

It is my belief that any new algorithm to implement an operator will not affect the final choice of allocation schemes. The choice of the k-dimensional projection with the stay-put modification allocation scheme was based on the data flow patterns rather than on the specific implementations of the operators. When a new method of implementing the reduction operator was found, this operator became independent of the allocation schemes and did not change is preference between two different allocation schemes.

Chapter 3

APL System

This chapter will present a proposed implementation of an APL system for the CHiP architecture. This description is not complete in that most of the evaluation algorithms for the operators will not be presented.

3.1 Language Modifications

Our proposed system is based on using a compiled version of APL rather than an interpreted version. The reason behind this is simply one of efficiency. A compiled version, along with modifications to the language such as declarations of the type of variables, will allow the system to avoid the normal type checking requirements found in an interpreted system. It will also allow the system to make more efficient use of the symbol table since both the number and names of the variables for a function will be known at compile time. This allows the system to make use of standard runtime stack routines.

In the final analysis we are willing to make changes to APL. These changes however should improve both the runtime efficiently of the system and still not alter any of the basic properties of APL.

3.2 Compiler Technology

The compiler technology used in implementing this system can be very simple. The major job of the compiler will be to produce the intermediate code used in executing the system, Very little optimization can be done at compile time. The two different types of optimization which can be done at compile time deal with idioms in of APL.

The first type of idioms that are to be examined are those that may be implemented as APL meta-operators. One example of this is the idiom $A[⍋ A]$, this can be converted into the meta-operator *sort* A. The meta-operator *sort* can be implemented more efficiently than $A[⍋ A]$. ⍋ implies that the sort be done and then converted into an index set to be used in subscripting the array. Rather than converting the final result back to an index set while computing the ⍋ operator, *sort* will produce the correct final array.

The second type of idioms that are to be examined are those which are equivalent. In a scalar system the expression $N\uparrow A+B$ will be converted into $(N\uparrow A)+N\uparrow B$ either by a compiler or during execution. In this implementation which is more efficient will depend on what the current allocations of A and B are. If A and B are in a strict allocation scheme then the expression $N\uparrow A + B$ should be evaluated as $A + B$ take zero IO cycles. If however either A or B is not in the strict allocation scheme then the expression $(N\uparrow A)+N\uparrow B$ should be evaluated as this will generally reduce the number of elements to moved in the system. (Note that with a good implementation of take that the number of elements to be moved can not increase even if N is greater than the size of the array A.)

3.3 Variable Allocation

There are three different variable entities for which allocation schemes must be given. These are **scalar variables**, **array variables**, and **constants**. It should be noted that a variable can bc either a scalar variable or an array variable, depending on the data structure which was last assigned to it.

The value of a scalar variable will be maintained in the controller processor as part of its symbol table entry for the variable. All computations that will use just scalar variables are to be done in the controller without involving the array processors. When an operation returns a scalar value, the value will be transferred from the processor array to the controller. Conversely when a scalar value is involved in an array operation, then the controller will broadcast its value to the processor array.

Array variables will be laid out in the processor according to the allocation scheme presented in the previous chapter. The allocation will be a k-dimensional projection onto the processor array, with the stay-put modification of tagging values with their indices, and delaying transfer of values to the correct processors.

Constants come in two flavors in APL. They may be either scalar constants or arrayed constants. Scalar constants will be maintained in both the controller processors and each processor in the processor array. Placing scalar constants in the processor array is an optimization step, and may be omitted if desired; in this case they would be treated the same as scalar variables. Arrayed constants can come about in two ways. First, through implicit operations in the APL code such as (1 2 3) which is a three element vector. Second, through constant folding operations such as evaluation of the expressions $\iota 10$ or $20\rho 3$ at compile time. Arrayed constants are stored in the same allocation structure as if they were variables: a k-dimensional projection.

3.4 Run Time Symbol Table

Two types of run time symbol tables will be maintained in the system, one for the controller and one for each element in the processor array. Not all elements of the symbol table will be associated with variables. Some will be intermediate results, the equivalent of registers in a scalar machine.

The controller symbol table will contain the value of the variable if the variable is currently a scalar, the number of dimensions of the variable and the shape of the variable. If the variables were not typed, then the type of the variable would be kept here as well. The entry for a variable may be directly accessed by the program, since a constant offset for the current procedure can be compiled into the system.

The symbol table for an array processor has associated with it the data elements contained in the processor. The program will be able to add new data elements, delete old data elements, sequentially access all data elements for a variable, access data elements for a variable using an index search pattern. Each data element will contain the actual value and the indices of a single array element.

The symbol table will be placed on the runtime stack of the processor. Constant offsets, either in the global or local symbol table areas, can be computed by the compiler since all variables are declared in the program. Additional space will be placed on the symbol table area to hold temporary results for by the compiler.

A special notation in both the controller and the processor symbol tables will provide for another optimization. When a scalar variable is broadcast to the array then it can be placed into the symbol table for the processors. The next time that the variable is to be used, if the value has not been changed then the broadcast step may be skipped.

3.5 Code Allocation

The code will be kept in all processors. Each processor will keep a separate program counter that will generally contain the same value at all times. The exception to this is that if a series of scalar operations are encountered in the controller, then the program counter in the array processors need not be updated until it is again required for evaluation of an operation.

The code would be compiled and stored as five-tuples consisting of the operation code, the result variable, and up to three source variables. The third source variable comes into play for those operations in which the dimension of the operation can be specified.

3.6 Execution Cycle

The same general execution cycle will be followed by all operations.

1. Look up the dimensions of the source variables in the symbol table.

2. Using the operand decide if the operation may be executed in the controller processor. If it can be then the operation is executed in the controller and the value stored in the controller symbol table. All program counters are increased and the next operation is started.

3. Inform the processor array of any special considerations about the source and result variables. Depending on the operation code this may include the

number of dimensions of a variable (scalar or array), the value of a variable (scalars) and the shape of the variable (arrays).

4. Set-up and start the processor array. This includes setting the phase (or phases) to be executed and, as needed, the interconnection structures. For some operations the controller will be passive while for others the controller will take an active part in evaluating the operation.

5. Obtain the result value if it is a scalar.

6. Increase the program counters in both the array and the controller.

3.7 Example Algorithms

To show how the execution model works, two example algorithms are presented here. Enough of the algorithm is presented to get the flavor of how it will work without doing a full implementation. More on the algorithms used in the simulation system may be found in Appendix A.

3.7.1 iota

Step	Controller	Array Processor
1	Read value of source variable from symbol table.	Delete all data values for the result.
2	Broadcast source value to processor array.	Read broadcast value.
3		Compute range of values to be mapped to this processor.
4	Update number of dimensions and the shape for the result variable	Create a data element for each value mapped to the processor and place in the symbol table.

3.7.2 Tree Reduction

Step	Controller	Array Processor
1	Broadcast the dimension to be reduced.	Read dimension to be reduced.
2	Compute the shape of the result, broadcast it to the array and store it in the symbol table.	Read the shape of the result.
3		Delete the data elements of the result from the processor.
4	Set interconnection structure to the binary n-cube.	Combine all data elements in the processor with the same result indices. The destination processor and the distance to the processor are packaged into the packet with the element.
	Repeat steps 5 and 6 *log*(Num. Processors) times.	
5		If current distance of packet equals maximum distance left then forward the packet to the next processor.
6		Read any forwarded packets combining the data elements with any internal elements having the same result indices.
7		Store the result data elements in the symbol table.

Chapter 4

Experimental Description

In the previous sections we have shown that it is indeed possible to implement a system that will evaluate APL programs in a nonshared memory model. At this point it is not possible to compare the proposed system to either existing systems or to a theoretical sequential system. A complete system has not yet been written. This section will instead describe a series of experiments run to examine some of the design decisions made in the proposed APL system.

The procedure for doing an experiment can be seen in Figure 4.1. APL programs were hand modified to produce an execution trace file when they were run on the PortaAPL system. A simulation program was written that takes as input the execution trace file. The trace is in the form of quadruples and is used as input to the simulation program. The simulation program then "executes" the APL program as if it were in a parallel environment. The simulation allows the data elements to move throughout the processor array and the controller in order to measure the amount of time taken in transmissions.

In this experiment we explored two different axes of the design space. The first axis looked at four different allocation strategies. The second axis of the design

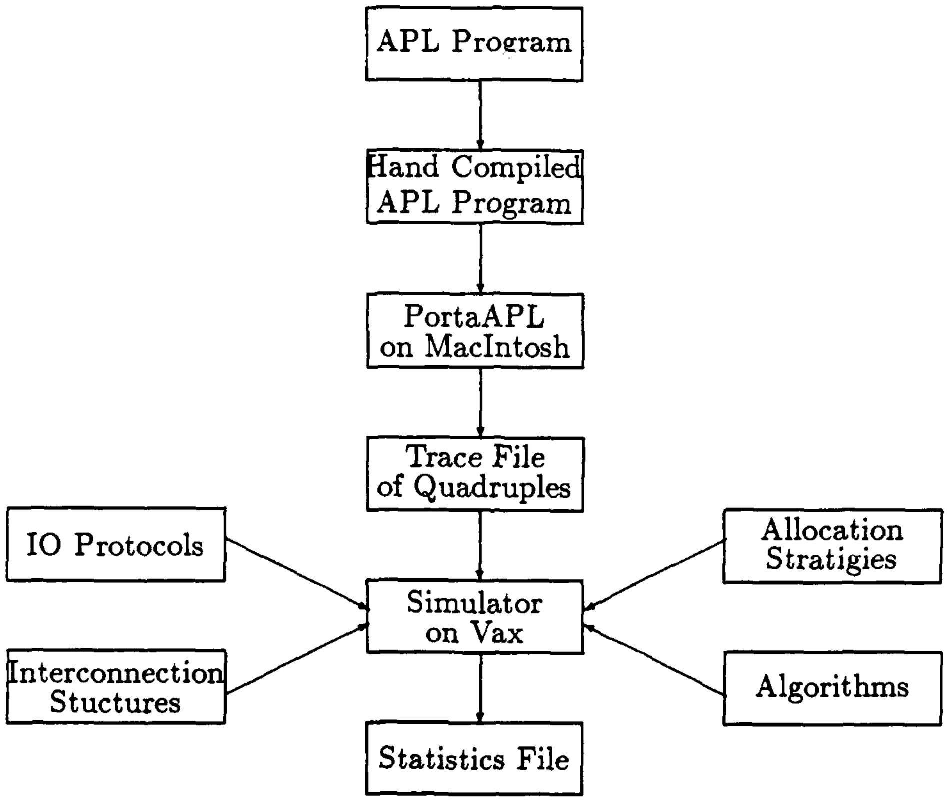

Figure 4.1: **Experimental Process** – This figure gives a flow chart of the experimental process. APL programs where hand modified to produce an execution trace file when run. The execution trace file was then used as input by the simulation program to find the total time. The simulation program is affected by the choice of Algorithms, Interconnection Structures, IO Protocals and Allocation Strategies.

space that was explored dealt with three IO strategies. These will be described after discussing the test programs and the machine model.

4.1 Program Set

A selection of seven APL programs was used to test the different options in the model. The programs ranged from character manipulation programs such as PRETTYPRINT, which reformats function definitions, to numeric simulation programs such as SORRB, which approximates a solution to $AX = Y$. The programs were either written by the author or taken from APL Quote Quad. The programs used are presented in Appendix C.

Each program was modified to output the quadruple code, along with any annotation needed for the specific operator, to a trace file during the execution of the program. The programs were run on an Apple MacIntosh using the PortaAPL system.

4.2 Machine Model

For the purposes of this experiment we modeled an 8x8 CHiP computer, modified so that a fixed interconnection structure was used. The interconnection structure was restricted to the binary n-cube network.[1] The controller processor is also included in the model.

Each processor has the following IO characteristics. In a unit time the processor is allowed to do a read cycle, a write cycle and an infinite amount of computation. The reasoning behind allowing an infinite amount of computation is that the amount

[1]This differs from the Cosmic Cube architecture in that an outside controller is available with the ability to do unit time broadcasts to the processor array and to control which code is currently executing in the processors.

of computation that can be done in the same amount of time as one read or write is large. There are sufficient ports to support the binary n-cube network on the CHiP. The processor has the ability to do simultaneous reads and/or writes on all ports. This allows us to do up to six reads and/or writes in a unit time. A read and write port are also provided to the controller processor and again one read and/or write is allowed in unit time. The ports contain an infinite number of write buffers with the ports doing the writes in first-in first-out order. The read buffers are only one unit in length and will be overwritten if not read.

The controller has the same characteristics as an array processor, *i.e.* in unit time it may perform a read, a write, and an infinite amount of computation. The controller, however, only has a single port pair to use. This port pair reads from and writes to a global bus. Each processor can also write on the global bus in a priority fashion (starvation prevention is up to the programmer). When a broadcast is done by the controller each array processor will decide under program control whether or not to read the value.

4.3 Allocation Space

Four different allocation schemes were examined during the experiment. These allocation schemes were:

- **Ravel Order Allocation** – The size of the array was computed and buckets are created to hold the elements of the array. Each element in the array then computes its ravel order index and, based on that, the bucket (or processor) in which it should be placed.

- **Gray Coded Ravel Order Allocation** - The same method of computing the bucket that a data value is in is used. The buckets do not however correspond directly to the processors, instead the buckets are mapped into the processors using the Gray code numbering of the processors.
- **Partial Gray Coded Ravel Order Allocation** — This allocation scheme uses an even more convoluted mapping scheme between the buckets and the processors. The processor array is numbered according to a 2-dimensional grid, the bucket associated with a processor is then the GrayCode(y)* NumberProcessorPerSize+GrayCode(x).
- **2-dimensional Projection Allocation** — Vectors are laid according to the same allocation structure as the Ravel Order Allocation using all processors. Matrices and arrays of higher dimension are laid out on a 2-dimensional grid by examining only the first and last dimensions of the array. Bucketing is performed independently in each dimension of the array to allow for the fixed number of processors.

All allocations: (1) Store scalar values in the controller, (2) bucket data values by collecting adjacent values in the array into a single bucket, and (3) use the same evaluation algorithms.

4.4 IO Protocol Space

Three different IO Protocols were examined during the experiment. These protocols were:

- **Single Data Value Packets** — Under this IO protocol each packet was individually sent between processors. This meant that only six data items could be written in one time unit.

- **Multiple Data Value Packets** — Under this IO protocol all data values that were to be written on a single port were packaged together and written as a single unit. The data values were then unpackaged after a read and either used locally or rerouted.

- **Broadcast Data Value Packets** — Under this IO protocol all data values that originated from a single processor were packaged together into a single packet. The reading processor then unpackaged every data item and, after using the ones needed locally, repackaged the data values according to the port on which they were to be written. This protocol was to take advantage of the broadcasts there were often done by a single processor during the course of evaluation.

4.5 Results

A total of twelve different points in the design space were tested. A summary of these points may be found in Table 4.1. The raw data in terms of total number of IO cycles may be found in Appendix B. A table was constructed in which the data was normalized by dividing the actual time for each entry in the table by the minimum execution time for a test program-data set pair across all experimental set-ups. The graphs in Figures 4.5–4.5 presented in this section are of the normalized data.

Figure 4.5 shows most of the interesting trends in the data. This graph shows the relative timings for the program PRIMTO, a program which computes the prime numbers using sieve of Eratoshenes. The data points for the multi-value IO protocol are approximately the same for the different allocation strategies. For the Single

Table 4.1: Table of Experiment Numbers

Experiment Number	Allocation Scheme	IO Protocol
1	Gray Coded Ravel Order	Single Data Item
2	Ravel Order	Single Data Item
3	Partial Gray Coded Order	Single Data Item
4	2-dimensional Projection	Single Data Item
5	Gray Coded Ravel Order	Multi-value Data Items
6	Ravel Order	Multi-value Data Items
7	Partial Gray Coded Order	Multi-value Data Items
8	2-dimensional Projection	Multi-value Data Items
9	Gray Coded Ravel Order	Broadcast Data Items
10	Ravel Order	Broadcast Data Items
11	Partial Gray Coded Order	Broadcast Data Items
12	2-dimensional Projection	Broadcast Data Items

Value IO protocol however the total times may be rather different for the different strategies. This is especially prevalent for the case of comparing the ravel order allocation with the 2-dimensional projection. The other graphs in Figures 4.5–4.5 show similar trends.

4.5.1 Allocation Discussion

One of the surprising pieces of data found in this experiment was the lack of dependency of the execution time on the allocation scheme used. In most cases a difference of less than ten percent was observed. There are three possible reasons why the data is so close together. First, the fixed interconnection structure may have affected the operator algorithm selection and may not have been the optimal interconnection structure for those algorithms used. Second, non-optimal algorithms may have been used in evaluating the interconnection structures due to simplifications or substitutions. An example of this is the operator ϵ which takes time in proportion to the size of the first variable rather than the minimum of the

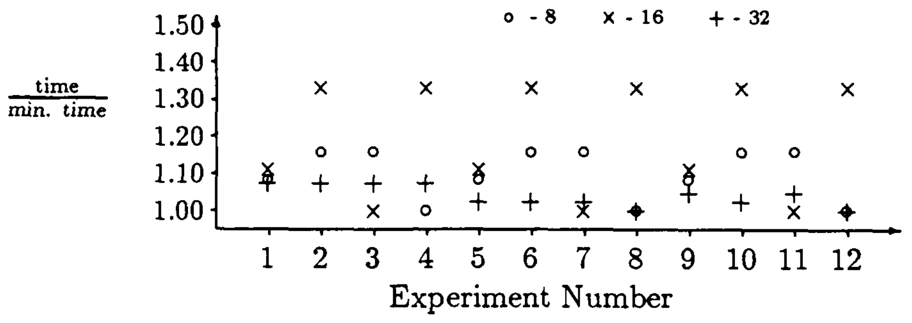

Figure 4.2: **SORRB Timings** – Relative timings for the SORRB program using three different data set sizes: an 8x8, a 16x16 and a 32x32 array. The SORRB program does a red-black SOR. It is an approximate method of solving $AX = Y$. The times are relative to the minimum for each data set size. The raw data is in Appendix B.

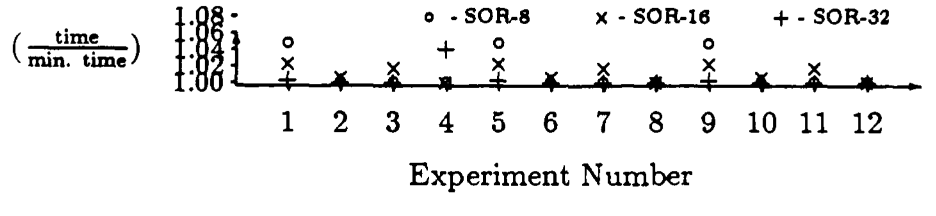

Figure 4.3: **SOR Timings** – Relative timings for the SOR program using three different data set sizes: an 8x8, a 16x16 and a 32x32 array. The SORRB program is an approximate method of solving $AX = Y$. The times are relative to the minimum for each data set size. The raw data is in Appendix B.

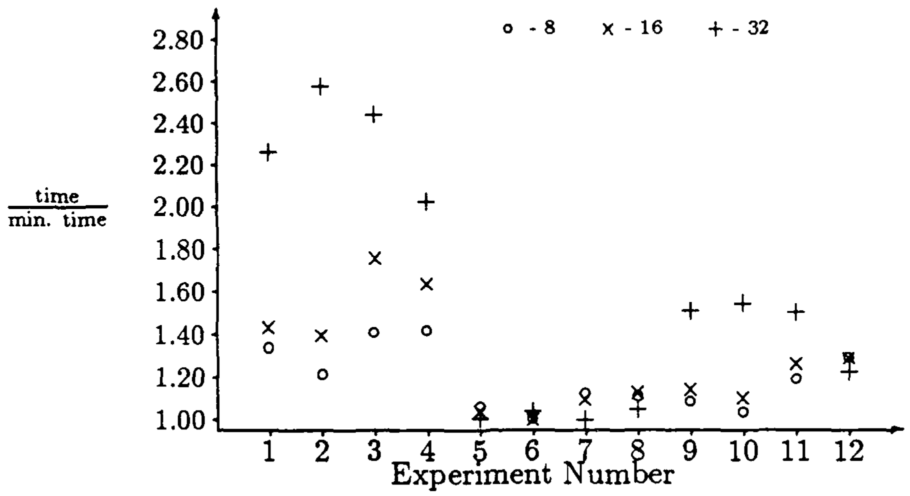

Figure 4.4: **GAUSS Timings** - Relative timings for the GAUSS program using three different data set sizes: an 8x8, a 16x16 and a 32x32 array. The GAUSS program solves $AX = Y$ by gaussian elimination. The times are relative to the minimum for each data set size. The raw data is in Appendix B.

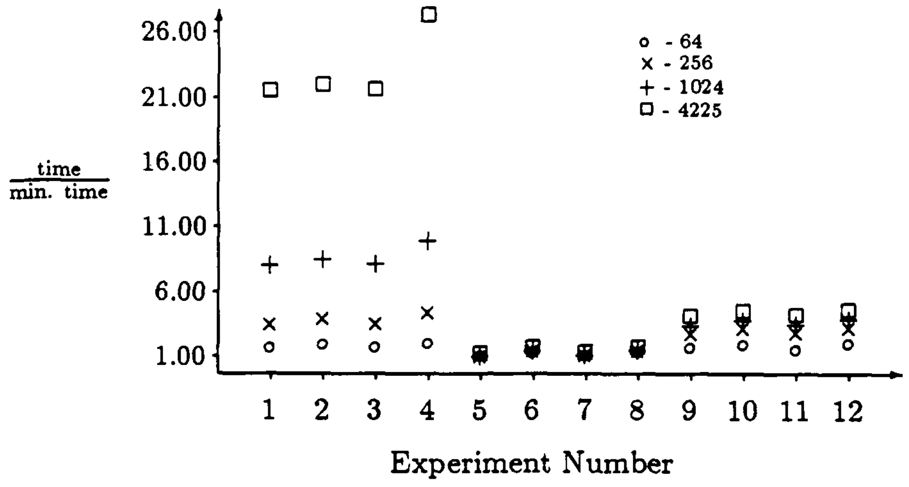

Figure 4.5: **PRIMTO Timings** – Relative timings for the PRIMTO program using four different inputs: 64, 256, 1024 and 4225. The PRIMTO program computes all prime numbers less than the input values using the sieve of Eratoshenes. The times are relative to the minimum for each data set size. The raw data is in Appendix B.

two variables. And third, the operators used may not have occurred in the expected distribution.

No allocation scheme can be said to have been the best. The Partial Gray Coded scheme was the worst of the allocation schemes. It was never better than one of the other three schemes and was generally worse than at least two of them.

Two things may have affected the results obtained from the experiment. First the programs used in the experiments tended to be numerical programs. These have a high usage of inner products which may tend to favor the 2-dimensional projection. Data set sizes used in these programs tended to be multiples of the number of processors which may have tilt the balance back. If the data set sizes for this numerical programs were to cover the range of possible values then a larger trend to the 2-dimensional projection allocation scheme would be expected to be

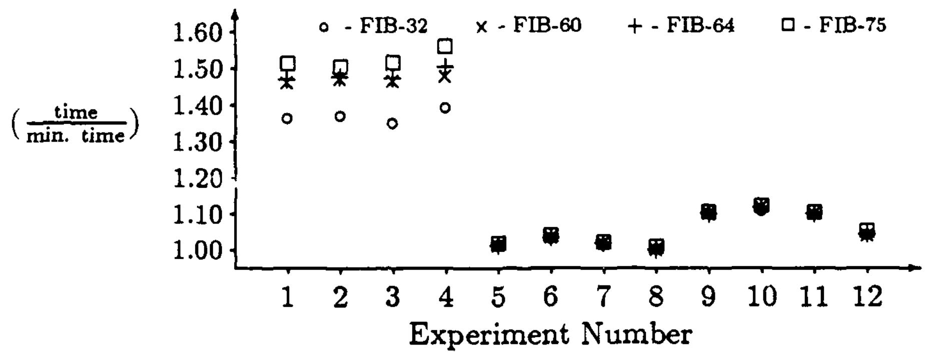

Figure 4.6: **PRETTYPRINT Timings** - Relative timings for the PRETTYPRINT program using four different display widths: 32, 60, 64 and 72 characters. The PRETTYPRINT program reformats an APL function to improve readability. The program FIB was the program to be reformatted for all data sets. The times are relative to the minimum for each data set size. The raw data is in Appendix B.

observed for these numerical programs.

The second thing that may have affected the results was the use of the binary n-cube as the interconnection structure. This was to the detriment of 2-dimensional projection as well. Operators such as scan can be more efficiently implemented if an explicit tree structure along each dimension is used rather than attempting to use any implicit tree structures found in the binary n-cube graph.

4.5.2 IO Protocol Discussion

Again as with the allocation data, most of the results were surprisingly close. The expected ordering was always maintained; that is the fastest protocol was Multi-value packets followed by Broadcast Packets and lastly Single-value Packets. A large difference between Multi-value packets and Single-value packets was expected due to the multiplicity of data elements in a processor. Many times, however, there was less difference than expected between the Multi-value packets and the Single-value packets. One possible conclusion from this is that altering the forwarding strategy will improve the single-value packets to the point that the difference may be minimal especially after one takes into account the time needed to pack and unpack the items. Two things can be done to improve the simple forwarding strategy used by the experimental system. The first would be to prioritize the order in which items are sent out the port; that is remove the simple FIFO ordering. Items would then be sent out in order of those that needed to travel the farthest rather than in the order generated. (These orders are often the opposite; those items sent out first are those which have the shortest distance to go.)

The biggest difference between any of the IO protocols occurred for the program PRIMTO. In analyzing the time spent on each operator it was found in the range of

70-90% of the time was spent doing subscripting for the lefthand side. Most of this time is spent in sending out the vector that is being subscripted, and this can be shortened considerably by packing together the values. A second way in which to reduce the difference is to improve the method of distributing the subscript values.

Chapter 5

Conclusions

The first conclusion that can be made is that an APL system can be written for a MIMD, nonshared memory machine. In addition it seems possible that such a system could be implemented in a reasonably efficient fashion. This system would greatly enhance the programmers view of MIMD nonshared memory systems. Programming MIMD machines through the use of systems such as the Cosmic Cube programming environment[17] is extremely difficult. The procedure is simplified somewhat by the Poker Parallel Programming environment[16], but it is still much more complicated both to understand what a program is doing and to program the code when one must write multiple interlocking programs. Thus we must be willing to pay some cost for the simplicity of using a shared memory model language.

The second conclusion is that *bigger is better*. A major reason why APL can be efficiently implemented on a nonshared memory computer is that there is no need to look for a global data flow pattern in the program. The data flow pattern is done once for each operator and the power of the operators is such that many operations may be done simultaneously. One place where this can be applied even more is to link together multiple APL operators and define meta-APL operators.

For instance the expression used to sort a vector of numbers is $V[⍋\, V]$. ⍋ produces a vector of indices based on the sorted sequence of the values, thus the sort and a permutation must be done to evaluate the expression. If one combines this into a single meta-operator then the value can be sorted along with the indices, the result of the operator has been computed without the need of the subscripting operation.

The final conclusion that we make is that the exact allocation scheme used by the data may not be as important as was originally believed. Given the use of the different operators in APL, those operators which favor a Ravel Order allocation may be almost balanced by those operators which favor the N-Dimensional Grid allocation.

Bibliography

[1] P. Abrams. *An APL Machine.* Technical Report SLAC Report #114, Stanford University, 1970.

[2] J. R. Allen and K. Kennedy. PFC: a Program to Convert Fortran to Parallel Form. In K. Hwang, editor, *Supercomputers: Design and Applications*, pages 186–205, IEEE Computer Society Press, 1985.

[3] H. W. Bingham. Content Analysis of APL Defined Functions. *APL '75 Conference Proceedings (Pisa, Italy)*, 60–66, June 1975.

[4] H. W. Bingham. Dynamic Usage of APL Primitive Functions. *APL '76 Conference Proceedings (Ottawa, Canada)*, 83–86, September 1976.

[5] T. Budd. *An APL Compiler.* Technical Report 81-17, University of Arizona, Tucson, Arizona, October 1981.

[6] T. Budd. An APL Compiler for a Vector Processor. *ACM Transactions on Programming Languages and Systems*, 297–313, July 1984.

[7] L. Gilman and A. Rose. *APL an Interactive Approach.* Wiley, 1976.

[8] L. Guibas and D. Wyatt. Compilation and Delayed Evaluation in APL. *5th ACM Symposium on Principles of Programming Languages*, 1–8, January 1978.

[9] R. F. Hobson. A Directly Executable Encoding for APL. *ACM Transactions on Programming Languages and Systems*, 314–332, July 1984.

[10] K. E. Iverson. *A Programming Language.* John Wiley and Sons, New York, 1962.

[11] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence Graph and Compiler Optimizations. In *Proc. of the 8th ACM Symp. on Principles of Programming Languages*, Williamsburg, Virginia, January 1981.

[12] S. Pakin. *APL\360 Reference Manual.* Science Research Associates, Inc., Chicago, Ill., 1968.

[13] H. Saal and Z. Weis. Some properties of APL programs. *APL '75 Conference Proceedings (Pisa, Italy)*, 292–275, June 1975.

[14] Charles L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[15] Lawrence Snyder. Introduction to the Configurable Highly Parallel Computer. *Computer*, 15(1):47–56, January 1982.

[16] Lawrence Snyder. Parallel Programming and the Poker Programming Environment. *Computer*, 17(7):27–36, July 1984.

[17] Wen-King Su, Reese Faucette, and Chuck Seitz. *C Programmer's Guide to the Cosmic Cube.* Technical Report 5203:TR:85, Computer Science Department, California Institute of Technology, September 1985.

Appendix A
Algorithm Summary

This appendix presents a short description of the APL operators as they are implemented in the simulation program. Some of these algorithms may not be optimal due to the fixed interconnection structure or implementation of a simpler version of the algorithm for an operator. Further elaboration on these algorithms may be found in the simulation program.

The most complete discussion of the implementation of the operators, including preliminary complexity analyses may currently be found only in the author's working notes.

In the description of the operators A will be used as the first variable for an operator and B will be used as the second variable for an operator.

A.1 Monadic Operators

Each processor execute the monadic function on the data elements contained within it.

A.2 Dyadic Operators

Each processor applies the function to combine the values with the same indices. If either variable is not in the strict allocation scheme then it is rearranged before invoking the dyadic operator.

A.3 Shape

The shape vector is broadcast to the processor array by the controller. Each processor keeps only those elements of the vector which are mapped to that processor

A.4 Index

The index variable is broadcast by the controller to the processor array. Each processor computes the range of elements that are mapped to that processor. The processor will then create the data elements mapped to it.

A.5 Reshape

The shape of the source variable is broadcast to the processor array. Each processor computes the ravel order index of the local elements. The reshape vector is then broadcast to the processor array and to the controller. Each processor uses the reshape vector to compute the new indices of each data element.

A.6 Take

Each processor modifies the index vector for the local elements according to the broadcast take vector. If additional elements need to be created each processor will create those elements which are mapped to it by the allocation scheme.

A.7 Drop

Each processor modifies the index vector for the local elements according to the broadcast take vector.

A.8 Assign

Each processor locally copies elements from the source variable to the destination variable.

A.9 Ravel

The shape of the source variable is broadcast to the processor array by the controller. Each processor computes the ravel order index of each local element.

A.10 Transpose

Each processor inverts the order of the indices associated with local elements.

A.11 Rotate

Each processor modifies the indices for the local elements according to the broadcast rotate vector and the shape of the source element.

A.12 Membership

The second variable is rotated through all processors using a ring. Each processor will then compare the values from the first argument and locally compute the result.

A.13 Outer Product

Using two rings each variable is passed around the array. When a processor reads in a data element that is to be used in a local computation then it is copied and saved. Only those result elements which are to be stored in the local processor are computed in that processor.

A.14 Catinate

Each processor modifies the indices of the second variable's local elements to directly follow the elements of the first variable. The offset is computed from the broadcast shape of the first variable.

A.15 Reduction

Linear

The linear arrangement of elements is induced from the indices of the data elements and the values are sent over the interconnection network and forwarded as needed to combine the data elements.

Tree

A binary tree is rooted at each node in the array. Those data elements which are used in computing the result(s) for the i-th processor are then sent up the i-th tree. As data elements meet at nodes of the tree they are combined and a single value is sent to the parent.

A.16 Compress

Using a binary tree the expression $A\times +\backslash A$ where A is the first argument is computed. Each element of the expansion is then broadcast to the processors in the array which need it. When an element of the expression is received the local indices of data elements are modified if necessary.

A.17 Left Subscripting

As each index variable is broadcast to the array the local data value indices are modified accordingly. The index variables are broadcast starting with the left index and moving right.

A.18 Right Subscripting

As each index variable is broadcast to the array the local data value indices are modified accordingly. The index variables are broadcast starting with the right index and moving left.

A.19 Inner Product

Both variables are reshaped into matrices and laid out in a grid allocation scheme. A modified version of the systolic matrix multiplication algorithm is then executed and the result is reshaped to its correct final form.

A.20 Grade Up and Grade Down

Grading is done by using a bitonic sorting algorithm. At each exchange step all elements (or a subset of elements) are exchanged and the top (bottom) i elements are kept, where i is the number of elements in the processor which the exchange was started.

Appendix B

Experiment Data

Two tables are presented in this appendix. The first table gives the raw data obtained from the simulations. The numbers represent the total count of IO cycles during the evaluation of the program. One IO cycle is a read, write, and execute cycle used as the basic unit of time in the simulation program.

	Gray Coded			Ravel Order			Partial Gray Coded			2-d Projection		
	One	All	Source	One	All	Source	One	All	Source	One	All	Source
SORRB-8	2798	2798	2798	2992	2992	2992	3000	3000	3000	2588	2588	2588
SORRB-16	4002	4000	4000	4790	4788	4788	3602	3600	3600	4790	4788	4788
SORRB-32	8980	8558	8756	8978	8556	8556	8978	8556	8754	8976	8356	8356
GAUSS-8	287	228	233	260	215	222	302	241	256	304	238	278
GAUSS-16	843	611	671	821	589	649	1035	645	743	964	663	758
GAUSS-32	3212	1428	2147	3662	1475	2194	3466	1421	2140	2876	1488	1742
SOR-8	8780	8780	8780	8383	8383	8383	8383	8383	8383	8383	8383	8383
SOR-16	17989	17989	17989	17692	17692	17692	17889	17889	17889	17592	17592	17592
SOR-32	36601	36601	36601	36501	36501	36501	36501	36501	36501	37987	36501	36501
DESIGN	1614	1171	1563	1649	1199	1600	1617	1175	1567	1763	1246	1652
PRIM.64	4178	4178	4178	4176	4176	4176	4177	4177	4177	4176	4176	4176
PRIM.257	65812	65812		65814	65814	65814	65812	65812	65812	65814	65814	65814
Pm2.64	145	91	144	167	112	167	151	96	132	178	115	174
Pm2.256	442	128	347	497	181	402	453	138	357	560	186	411
Pm2.1024	1826	227	761	1924	326	863	1847	248	783	2248	333	878
Pm2.4225	7654	359	1386	7802	517	1543	7687	393	1419	9721	519	1561
FIB.32	640	476	516	643	484	520	633	476	514	653	470	489
FIB.60	687	476	517	692	486	526	689	479	518	695	470	489
FIB.64	691	476	518	694	486	525	693	479	518	707	470	490
FIB.75	707	474	515	703	485	523	708	476	515	729	470	490

This table gives the normalized data for the experiments run. The execution times for each data set were divided by the minimum execution time for the data set. This table has the same data that is presented in Figures 4.5–4.5.

	Gray Coded			Ravel Order			Partial Gray Coded			2-d Projection		
	One	All	Source	One	All	Source	One	All	Source	One	All	Source
SORRB-8	1.08	1.08	1.08	1.16	1.16	1.16	1.16	1.16	1.16	1.00	1.00	1.00
SORRB-16	1.11	1.11	1.11	1.33	1.33	1.33	1.00	1.00	1.00	1.33	1.33	1.33
SORRB-32	1.07	1.02	1.05	1.07	1.02	1.02	1.07	1.02	1.05	1.07	1.00	1.00
GAUSS-8	1.33	1.06	1.08	1.21	1.00	1.03	1.40	1.12	1.19	1.41	1.11	1.29
GAUSS-16	1.43	1.04	1.14	1.39	1.00	1.10	1.76	1.10	1.26	1.64	1.13	1.29
GAUSS-32	2.26	1.00	1.51	2.58	1.04	1.54	2.44	1.00	1.51	2.02	1.05	1.23
SOR-8	1.05	1.05	1.05	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
SOR-16	1.02	1.02	1.02	1.01	1.01	1.01	1.02	1.02	1.02	1.00	1.00	1.00
SOR-32	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.04	1.00	1.00
PRIM.64	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
PRIM.257	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Pm2.64	1.59	1.00	1.58	1.84	1.23	1.84	1.66	1.05	1.45	1.96	1.26	1.91
Pm2.256	3.45	1.00	2.71	3.88	1.41	3.14	3.54	1.08	2.79	4.38	1.45	3.21
Pm2.1024	8.04	1.00	3.35	8.48	1.44	3.80	8.14	1.09	3.45	9.90	1.47	3.87
Pm2.4225	21.32	1.00	3.86	21.73	1.44	4.30	21.41	1.09	3.95	27.08	1.45	4.35
FIB.32	1.36	1.01	1.10	1.37	1.03	1.11	1.35	1.01	1.09	1.39	1.00	1.04
FIB.60	1.46	1.01	1.10	1.47	1.03	1.12	1.47	1.02	1.10	1.48	1.00	1.04
FIB.64	1.47	1.01	1.10	1.48	1.03	1.12	1.47	1.02	1.10	1.50	1.00	1.04
FIB.75	1.50	1.01	1.10	1.50	1.03	1.11	1.51	1.01	1.10	1.55	1.00	1.04

Appendix C

APL Program Listings

C.1 SOR

The program SOR was written by the author. The program computes an approximate solution for $AX = Y$. The technique uses the partial solution for this iteration in solving for the rest of the current iteration.

```
X←A SOR Y;I;J;XOLD;XC;DIAG;N;NA
X←(N← ρY)ρ3
DIAG← 1 1 ⍉ A
NA←A×(ιN)∘ .≠ ιN
J←1
LO:XOLD←X
I←1
L1:XC←(Y[I]-NA[I;]+.×X)÷DIAG[I]
X[I]←(XC×W)+XOLD[I]×1-W
→(N≥I←I+1)/L1
→((TOL<⌈/X-XOLD)∧(MAXITER>J←J+1))/LO
```

C.2 SORRB

The program SORRB was written by the author. The program computes an approximate solution for $AX = Y$. The technique used is similar to the SOR program.

However the results points are labeled red and black such that no two red points or black points affect each other. This allows the system to solve each iteration in two steps. The amount of parallelism in the system is greater than for the straight SOR program with a corresponding restriction on the problems which can be solved.

```
X←A SORRB Y;RDIAG;BDIAG;RA;BA;RX;BX;RY;BY;XOLD;XC;N
N←(ρY)÷2
RDIAG←(1 1)⍉ (N,N)↑A
RA←(N,-N)↑A
BDIAG←(1 1)⍉ (N,N)↓A
BA←(N,-N)↓A
RY←N↑Y
BY←N↓Y
RX←Nρ3
BX←Nρ3
J←1
LO:
XOLD←RX,BX
XC←(RY-RA+.×BX)÷RDIAG
RX←(XC×W)+RX×1-W
XC←(BY-BA+.×RX)÷BDIAG
BX←(XC×W)+BX×1-W
'J= ';J;'  ';⌈/|XOLD-RX,BX
→((TOL<⌈/|XOLD-RX,BX)∧(MAXITER>J←J+1))/LO
X←RX,BX
```

C.3 GAUSS

The program GAUSS was written by the author. The program computes an exact solution for $AX = Y$, using gaussian elimination.

```
X←A GAUSS Y; I;N;J;V;B
I←1
N← ρY
A←A,Y
L1:→(I=J←(<\0≠(I-1)↓A[;I])/(I-1)↓ιN)/COMPUTE
→(0=ρJ)/ERRORS
```

```
V←A[I;]
A[I;]←A[J;]
A[J;]←V
COMPUTE:B←(N,N+1)ρA[I;]
B[I;]←0
A←A-B×(⍉ (⌽ρA)ρA[;I])÷A[I;I]
→(N≥I←I+1)/L1
X←A[;I]÷(1,1)⍉ A
→0
ERRORS:
```

C.4 PRIM

This program to compute prime numbers is taken from a lecture by . The lecture may be found in APL Quote Quad Vol. 16, No. 2, December 1985.

A Prime number is one that is not found in the multiplication table of integers, starting with 2. The program PRIM is a direct implementation of this statement.

```
Z ←PRIM N;Q
Z ←(~Q∈Q∘.×Q)/Q←1↓⍳N
```

C.5 PRIMTO

This program to compute prime numbers is taken from a lecture by . The lecture may be found in APL Quote Quad Vol. 16, No. 2, December 1985.

The function PRIMTO finds all prime numbers up to some given number. It uses successive reshapings to efficiently implement the sieve of Eratosthenes.

```
Z ←PRIMTO N;S;M;NP;R
S←N ⋆0.5
M←(N,1)ρ ~N↑1
NP←1
L1:
   NP←NP + M[2;]ι1
   →(NP>S)/L2
   M←((R←⌈N÷NP), NP)ρM
   M[1↓ιR; NP]←0
   →L1
   L2:
   Z←(NρM)/ιN
```

C.6 PRETTYPRINT

The PRETTYPRINT program was taken from APL Quote Quad Vol. 16, No. 2, December 1985. The program was written by Ross Bettinger.

The program reformats a function definition exdenting the labels and separating the comments from the code. It does this without the use of any loops making a perfect example of what APL programs can do.

```
X←PW PRETTYPRINT FCN;COLON;LABCOL;COMLIN;LABROW;LABMAT;LINE;WRAP;Z
⍝  PURPOSE: FORMAT   CR OF FCN INTO CHAR MATRIX FOR PRINTTING
⍝
⍝ ACKNOWLEDGEMENT: THE IDEA FOR FORMATTING APL FCNS IN THIS MANNER IS
⍝    DUE TO ADRIAN SMITH, AS PRESENTED IN HIS EXCELLENT BOOK,
⍝    apl-a design handbook for commerical systems
⍝    PUBLISHED BY JOHN WILEY AND SONS, 1982.
⍝
⍝ FCN ←→ NAME OF FCN TO BE FORMATTED
⍝ PW ←→ PRINT WIDTH OF FORMATTED FNC (≥ 40 COLS)
⎕IO←1
FCN←  CR FCN
PW←40⌈PW
⍝  ADD '∇' TO HEADER, BELOW FCN
FCN←(1 2+ρFCN)↑FCN
FCN[1;ι1↓ρFCN]←¯2↓'∇ ',FCN[1;]
FCN[1↑ρFCN;1]←'∇'
⍝ CREATE LINE NUMBERS
LINE←(⍕  (Z,1)ριZ←¯2+1↑ρFCN),']'
LINE←'[',[2](+/' '=LINE)⌽LINE
LINE←' ',[1]LINE,[1]' '
⍝  ADJUST PW TO REFLECT LINE NO'S SPACE REQMTS
```

```
PW←PW-1↓ρLINE
⍝  FIND COMMENT LINES TO BE ROTATED LEFT 1 COL
COMLIN←'⍝'=FCN[;1]
⍝ INSERT 'α' TO SET APART COMMANDS
Z←¯.5+((COMLIN>0,¯1↓COMLIN),COMLIN<0,¯1↓COMLIN)/(2×ρCOMLIN)ριρCOMLIN
Z←(((ρCOMLIN)ρ1),(ρZ)ρ0)[⍋(ιρCOMLIN),Z]
FCN←Z\[1]FCN
FCN[(' '=FCN[;1])/ι1↑ρFCN;1]←'α'
COMLIN←'α'=FCN[;1]
⍝  EXPAND LINE NO'S TO MATCH EXPANDED FCN
LINE←Z\[1]LINE
⍝  FIND ROTATIONS FOR LABEL EXDENTATION
LABCOL←(ȻOMLIN)×+/Z←∨\⌽<\':'=FCN
⍝  SET UP MATRIX CONTAINING LINE LABELS ONLY
LABMAT←(0<LABCOL)/[1]FCN[;ι⌈/LABCOL]
LABMAT←(ρLABMAT)ρZ\(Z←,(ρLABMAT)↑(0<LABCOL)/[1]⌽Z)/,LABMA
⍝  GET INDICES OF LINE LABELS
LABMAT←LABMAT∊':ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopz0123456789∆'
LABROW←(LABCOL[Z]=+/LABMAT)/Z←(0<LABCOL)/ιρLABCOL
⍝  EXDENT LINE LABELS ONLY
Z←(1↑ρFCN)ρ0
Z[LABROW]←1
⍝  IF NO LINE LABELS, USE COMENT COLUMN
COLON←1+⌈/LABCOL←LABCOL×Z
⍝  RESHAPE FCN SO THAT 0←→ PW|1↓ρFCN
WRAP←⌈(1↓ρFCN)÷PW
FCN←((WRAP×1↑ρFCN),PW)ρ((1↑ρFCN),PW×WRAP)↑FCN
⍝  CREATE EXPANSION VECTO TO MATCH WRAPPED FCN
Z←(1↑ρFCN)ρWRAP↑1
⍝  EXDENT COMMENTS, LABELS IN RESHAPED FCN
FCN←((Z\LABCOL)+Z\COMLIN)⌽((1↑ρFCN),-COLON+1↓ρFCN)↑FCN
⍝  PUT LINE NUMBERS ONTO FCN
FCN←(Z\[1]LINE),[2]FCN
⍝  SAVE COL NO. OF LABEL COLONS, '⍝k CHARS
COLON←''ρCOLON+1↓ρLINE
⍝  OMIT BLANK ROWS IN FCN
FCN←(FCN∨.≠' ')/[1]FCN
⍝  SKIP ONLY ONE LINE BTWN COMMENTS
X←(Ƶ∧1⌽Z←'α'=FCN[;COLON+1])/[1]FCN
⍝  REPLACE 'α' COMMENT LINE SEPARATOR WITH BLANK
Z←'α'=X[;COLON+1]
X[Z/ι1↑ρZ;COLON+1]←' '
```

C.7 DESIGN

From APL Quote Quad Vol. 15, No. 3, March 1985. The program was written by Carina Heiselbetz.

The program produces a tower chart (skyscraper diagram) for contingency tables.

```
Z←DESIGN X;XY;N;M;Z1;Z2;Z3;Y;I;J
Z←(N←¯1↓ρX) FIELD M←1↓ρX
XX←100×X÷⌈/,X
Z1←0⌈Z1←¯4+⌈.1×⌈/,XX
Z←((Z1,1↓ρZ)ρ' '),[1]Z
J←I←1
LOOP:
→(XX[I;J]<0)/L1
Z2←Z1+¯1+7×I
Z3←(7×N-I+1)+17×J
→(XX[I;J]<5)/L2
Y←1+⌊.1×¯5+XX[I;J]
Z[Z2-Y;Z3+¯1+⍳4]←1 4ρ'/¯¯/'
Z[Z2;Z3+3]←'/'
Z[Z2-⍳Y;Z3+4]←'|'
Z[Z2-⍳Y-1;Z3+3]←' '
Z[Z2+1-⍳Y;Z3+¯2+⍳4]←(Y,4)ρ'|**|'
L1:
→(M≥J←J+1)/LOOP
→(N≥I←I+J←I)/LOOP
→END
L2:
Z←[Z2;Z3+¯1+⍳5]←'/___/'
Z[Z2-1;Z3+⍳5]←'/¯¯¯/'
→L1
END:
Z←Z,[1]' '
Z[¯1↓ρZ;9+⍳17×M]←(17×M)↑,(⍉ 1 6ρ'ABCDEF'),6 16ρ' '
I←1
L5:
Z[Z1+5+7×I-1;1+7×N-I]←⍕ I
→(N≥I←I+1)/L5
```

```
A←N FIELD M;I;J
A←((7×N)+1,17×M)ρ' '
J←1+I←0
L1:
A[1+7×N-I;(I×7)+ιM×17]←'_'
→←(N≥I←I+1)/L1
L2:
A[(7×N)+2-J;J,J+17×ιM]←'/'
→(7×N)≥J←J+1/L2
```