CORNELL UNIVERSITY

ITHACA, NEW YORK

SCHOOL OF ELECTRICAL ENGINEERING

THE LANGUAGE PARALLEL PASCAL AND OTHER ASPECTS OF THE MASSIVELY PARALLEL PROCESSOR

Final Report for the Period
January 1980 to December 1982

Anthony P. Reeves*
John D. Bruner**
School of Electrical Engineering
Cornell University
Phillips Hall
Ithaca, NY 14853

Work Supported by NASA Grant 5-3

* Most of this work was conducted while the authors were with the School of Electrical Engineering, Purdue University.

** John Bruner is now with the Lawrence Livermore Laboratory.

Abstract

This is the final report for NASA grant NAG 5-3 which is concerned with high-level language, computer architecture and algorithms for the Massively Parallel Processor (MPP). Previous work for this grant is described in Purdue technical reports TR-EE 80-32 and TR-EE 81-45. In this report only the recent up-dates to previous work and results of new work are given; previous work which is not described in detail here is mentioned in section one.

The main effort of the research has been to design a high level language for the MPP. This language, called Parallel Pascal, is described in detail in this report. Report sections include a description of the Language design, a description of the intermediate Language, Parallel P-Code, and details for the MPP implementation. Appendices give formal descriptions of Parallel Pascal and Parallel P-Code. A compiler has been developed which converts programs in Parallel Pascal into the intermediate Parallel P-Code language; the code generator to complete the compiler for the MPP is being developed independently by CSC for NASA. A Parallel Pascal to Pascal translator has also been developed. This allows Parallel Pascal programs to be developed and run on conventional computers without the need for direct access to the MPP.

In related work the architecture design for a VLSI version of the MPP is completed with a description of fault tolerant interconnection networks. In another section the memory arrangement aspects of the MPP are discussed and a survey of other high level languages is given in an appendix.

CONTENTS

1. Introduction

This is the third and final report for grant NAG 5-3. The other two interim reports TR-EE 82-32 [1] and TR-EE-81-45 [2] contain much information which is only briefly summarized in this report. The reader is referred to these reports for a complete detailed description of the work conducted for this grant.

The main concentration of the research effort has been directed towards the development of a high level language for parallel processors in general and the MPP in particular. Such a language, called Parallel Pascal, has been developed and is described in detail in this report.

In addition to this work, we have also conducted research into advanced architectures for Parallel processors such as the MPP and have programmed some algorithms for the MPP. In the remainder of this introductory section the work in languages, architectures and algorithms is briefly summarized and then an outline for the remainder of this report is given.

1.1 High Level Languages

There is an obvious need for the availability of a high level language for programming parallel processors such as the MPP. In our research in this area we have considered languages based on APL, Fortran and Pascal. The majority of the research has been devoted to the development of a language called Parallel Pascal. A specification for a parallel APL is given in [2], section 4 and the specification for a Parallel Fortran, which is relatively simple to implement, is given in [2], section 3. In Appendix 6 a general discussion is presented on the high level languages which have been developed for other parallel processors. This discussion includes their relevance and shortcomings with respect to parallel matrix processors including the MPP.

In this report section 2 contains the recent developments to the language since reports [1] and [2]. The I/O section of the language is specified here, and implementation restrictions on the compiler for MPP are described. Methods of programming the I/O for large size images with the implemented MPP language are also outlined in this section. In section 3, the design of Parallel Pascal is presented. This section starts with a discussion of the design goals of the language and continues to introduce the features which have been added to conventional Pascal in a logical, step by step manner. In Appendix A a formal specification of the Parallel Pascal language is given including a complete grammar.

A large part of the research effort was directed towards the specification of an intermediate compiler language called Parallel P-code which was developed from the P-Code intermediate language used in many Pascal compilers. The design of this language is presented in section 4 and a more formal language specification is given in Appendix B. This language may be used for the compilers of languages other than Parallel Pascal such as Parallel Fortran. A compiler which compiles Parallel Pascal into this intermediate language has been developed as part of the work for this grant NAG 5-3.

A preliminary description of Parallel P-Code was given in [2] section 2.4, extensive revisions have been made to the language since then. These revisions were caused by the complexity of the new language features and the different memory systems which the data may be mapped onto. The resulting language is at a higher, more symbolic level than conventional P-Code which gives the code generator more flexibility for optimization and allocation of memory.

A Parallel Pascal translator has also been developed which is described in [2] sections 2.2 and 2.3. This translator allows programs written in Parallel Pascal to be compiled and run on conventional computers which have a Pascal compiler. This is a very important tool which enables Parallel Pascal

programs to be developed, debugged and tested without the MPP. Consequently, programs may be developed on the users local computer at the users convenience even before the MPP hardware is available.

1.2 Computer Architecture

We have considered several architecture alternatives and extensions to the basic MPP design. The first feature which we considered important is a hardware bit-counting mechanism which can rapidly count the number of bits in a bit-plane. This mechanism was considered to be important for algorithms involving global feature extraction. A hardware bit counter design is presented in [3] where it is shown that a very large speed improvement over current MPP bit-counting methods can be achieved at a small cost. Algorithms where bit counting is important are also discussed in [3]. In reference [4], which is also in Appendix B of [2], algorithms are described for real-time image tracking and it is shown that the MPP with the bit counting hardware could implement these algorithms in real-time.

The construction of an MPP like array using VLSI technology components has been considered in [1], section 4. A three chip set is proposed consisting of a dense PE ALU chip, a local memory chip and a fault tolerant interconnection chip. The ALU chip is designed for optimal bit-serial multiplication speed which is much faster than the MPP design; it also has a table-look-up capability which is not available on the MPP. An extended interconnection scheme, called the two-dimensional perfect shuffle, is considered which overcomes most of the problems of the mesh interconnection scheme used on the MPP; but the implementation cost is very high.

In this report, section 6, a more detailed design of the fault tolerant interconnection ship is presented. It is shown that a very large amount of fault tolerance in the mesh connected array can be achieved at a very reasonable cost. This means that the construction of arrays much larger than the current

MPP size of 128x128 can be considered in the future. A second possibility for future consideration is to implement the whole PE array on a single silicon slice consisting of many interconnected chips. Reconfiguration to avoid bad parts of the chips can be done in software once the slide has been completely fabricated. Furthermore, if additional faults occur in the PE array at a later time, then software reconfigurations can be used again to avoid these new faults.

1.3 Algorithms

The design of the high level language and the computer architecture should both be algorithm driven. We have considered typical algorithm implementations at both high and low levels.

The MPP does not have a table-look-up facility, in [5] and also in [1] section 3, an efficient mechanism for implementing arbitrary functions on a bit-serial computer architecture is described and a function compiler for the MPP has been developed. The specification of the function is input to this function compiler which generates an optimized subroutine in bit-level assembler code for implementing the function. Any arbitrary function may be implemented; however, the number of instructions generated by the compiler increases exponentially with the number of bits of the function arguments. This technique is most suitable for arguments of 8-bits and less and for functions which cannot be implemented by a very simple direct method.

Various different algorithms have been programmed in Parallel and are presented in [1] and [2]. In [1] section 2.3, programs are given for PE address generation, image rotation, bilinear image resampling, maximum likelihood classification, convolution, histogram generation and isodata clustering. In [2] section 2.2.5, programs are given for manipulating large arrays on the 128x128 PE array of the MPP.

In reference [2], some basic parallel algorithms are discussed. Algorithms for local window operations including local sorting and local median filtering are described in section 5 of [2], these algorithms have also been published [4]. In Appendix B of [2], algorithms for rapid sequential frame registration, enhancement and feature extraction are described.

In this report sections 2 and 3 and Appendix A deal with the final specification of the Parallel Pascal Language. Section 4 and Appendix B deal with the intermediate Parallel P-Code language. Section 5 is concerned with memory management issues, i.e. methods to overcome the limitations caused by the small local memory size on the MPP. Finally, section 6 deals with the design of a VLSI interconnection chip which completes the design for a future VLSI MPP-like architecture. The majority of the effort during this last period in grant NAG 5-3 has been directed towards finalizing the design of Parallel Pascal and to fully consider the constraints of the MPP implementation. The work on the Parallel Pascal compiler which generates the Parallel P-Code has been completed.

1.4 References

1. A.P. Reeves and J.D. Bruner, "High Level Language Specification and Efficient Function Implementation for the Massively Parallel Processor", Purdue Technical Report TR-EE 80-32, July 1980.

2. A.P. Reeves, J.D. Bruner and T.M. Brewer, "High Level Languages for the Massively Parallel Processor", Purdue Technical Report TR-EE 81-45, Oct. 1981.

3. A.P. Reeves, "On Efficient Global Extraction Methods for Parallel Processors", Computer Graphics and Image Processing, Vol. 14, pp. 159-169, 1980.

4. A.P. Reeves, "The Local Median and Other Window Operations on SIMD Computers", Computer Graphics and Image Processing, Vol. 19, pp. 165-178, 1982.

5. A.P. Reeves and J.D. Bruner, "Efficient Function Implementation for Bit-Serial Parallel Processors", IEEE Trans. on Computers, Vol. C-29, No. 9, pp. 841-844, September 1980.

2. Parallel Pascal Update

The current specification of Parallel Pascal is very similar to the specification given in TR-EE 81-45. The main changes, made to control structures array indexing and I/O, are outlined below in section 2.1. A complete specification is given in Appendix A.

The restrictions which will be made to the initial MPP compiler have been determined and these are described in section 2.2. Several examples of performing I/O on the MPP are outlined section 2.3.

2.1 Revisions and I/O Specification

The only control construct which can have an array control variable is the where-do-otherwise construct. This is similar to the if-then-else construct with the following differences:

1. The control expression may have an array data type
2. All the targets of assignments must be conformable with the control expression, i.e., they must be either a similar sized array or a scalar.
3. Both the do and the otherwise sections will be executed in sequence; the do first and then the otherwise with the complement of the condition.

The where structure involves conditional assignment rather than conditional evaluation. Where structures may be nested with other where structures and with other convential control structures.

In report TR-EE 81-45 the concepts of subrange constant and subrange indexing were introduced. For example, the expression a[11..20] specifies a subvector of a consisting of elements a[11]through a[20]. This feature has now been extended to include an offset expression. For example, a[5 @ 11..20] adds the offset 5 to the constant subrange and results in the elements a[16] through a[25]. This offset has a similar effect to the shift function but is notationally more convenient in some cases and may be used on the left side of an assignment. The offset

may be specified by an expression whereas the subrange must be a constant. The syntax given for subrange indexing has been changed slightly since the grammer using the old syntax would no longer be of type LL(1) which is a requirement for the simple parsing of Pascal.

The I/O specification for Parallel Pascal will be the same as that for conventional Pascal. Parallel array I/O will be done with files declared to have Parallel arrays as basic elements. Special techniques for dealing with very large arrays and for reformatting array data are outlined in section 2.3.

The names of the standard reduction functions have been changed; however, these functions are still defined in the same way. The old and new names for these functions are given below:

Old Name	New Name	Function
asum	sum	sum
aprod	prod	product
aand	all	AND
aor	any	OR
amax	max	maximum
amin	min	minimum

2.2 The MPP Parallel Pascal

The initial Parallel Pascal to be implemented on the MPP will have several basic restrictions. Some of these restrictions may be removed with subsequent versions of the compiler.

The most fundamental restriction is that the last two dimensions of any parallel array must be 128 x 128 (or the last dimension must be 16384). It was decided that to not hide the machine architecture from the programmer in this way was necessary, at least in the early programming of the MPP, to ensure that well structured efficient programs are developed. The local memory is very limited on the MPP and the processing efficiency is greatly reduced if arrays are used which are not multiples of 128 x 128. For effecitve programming the user must be aware of these characteristics; in some cases different array sizes may dictate different programming strategies to efficiently implement the same function. A future compiler may contain some built in strategies for arbitrary sized arrays, but these will not be optimal for all cases. Techniques for dealing with large arrays in MPP Parallel Pascal are disucssed in section 2.3

Any arrays having the last two dimension other than 128 x 128 will be stored in the staging buffer. Only subarrays having the last two dimensions 128 x 128 can be directly processed; smaller subarrays may be "read" or "written" by assignment statements.

A second restriction is that parallel arrays cannot contain pointers or records. It is possible for a parallel array to contain records without variant parts; however, a record of parrallel arrays is probably a better data structure to use in this case. Pointers are not allowed since they would, in general, point to a different memory system and would be very difficult to manage.

Finally, there may be some minor restrictions to procedures or program blocks which contain parallel expressions. These will be a feature of the code generator which may be removed at a later date and are outside the scope of this report.

2.3 Large Array Processing

In this section the processing of arrays larger than 128 x 128 is considered. Also a mechanizm for using the reformatting features of the staging buffer is described. To aid clarity all declarations of parallel arrays which are to be located in the staging buffer rather than the PE array unit will be specified by buffer array rather than parallel array. Only single band arrays will be described, however multispectral data may be easily accomodated by defining the arrays to have one more dimension.

Large arrays will be considered to be of two types (a) arrays which will fit in the staging buffer and (b) arrays which are too large for the staging buffer. Type (a) arrays are considered first.

2.3 1 Whole arrays in the MPP

For arrays with the last dimensions being 128 x 128 the following program example is typical for reading an array

```
Type
     pixel  = 0..255;
     MPPA   = parallel array [0..127,0..127] of pixel;
Var
     f : file of MPPA;
     a:  MPPA;
Begin
     reset (f);
     read  (f,a);
       .
       .
       .
```

For arrays larger than 128 x 128 that are to be accessed in 128 x 128 chunks the following scheme may be used (for a 384 x 640 array in this case).

```
Type
     pixel = 0..255;
     MPPA  = parallel array [0..127,0..127] of pixel;
     BUFA  = buffer array  [0..383, 0..639] of pixel;
Var
     bf:  file of BUFA;
      a:  MPPA;
      b:  BUFA;
Begin
      reset (bf);
      read  (bf,b);
       .
       .
       .
      a: = b[0..127,256 @ 0..127];
       .
       .
       .
```

A second alternative, if the array is to be processed as a single entity in the MPP (as outlined in section 2.5 of TR-EE 81-45), is to consider the array to have dimensions 3 x 5 x 128 x 128 as shown below.

```
Type
     pixel = 0..255;
     MPPL  = parallel array [1..3, 1..5, 0..127,0..127] of pixel;
Var
     fc:     file of MPPL;
     c :     MPPL;
Begin
     reset (fc);
     read  (fc, c) ;
       .
       .
       .
```

2. 3.2 Partial arrays in the MPP.

When the array is too large to fit into the MPP then it must be stored on the disk in convenient sized chunks. The formatting of the data on the disk could and should be done by a back end processor since the processing requirement of this task is very low and would be a waste of time on the MPP. Alternatively the reformatting could be done by the MPP at the beginning of the program.

The data, therefore, is in the form of a file of chunks; for simplicity we will consider each chunk to have the last dimensions 128 x 128 although they could also be a multiple of 128. Random access of these chunks is possible but the seek time of conventional disk systems will make this scheme very slow. A "seek" function in Pascal would be very useful for random file access and could easily be implemented in the Parallel Pascal compiler. A faster mode of operation, which is adequate for most applications, is to spool the data through the MPP. In this case the sequence of disk accesses is known and the data may be arranged on the disk to minimize seek time.

The spooling system could be written as a set of Parallel Pascal library functions. The following functions would be required: resets, open a spooling file; reads, read the next block; writes, write the next block; and closes, close the spool file. Each reads and writes operation will access the next sequential block of the large array file. Four modes of data access have been considered and are illustrated in Fig. 2.3.1. Each mode is useful for a particular class of algorithm.

In mode zero each block is overlapped with the previous one by a specified amount. In this way some edge effects, caused by sequential block accessing, can be ignored.

In the simple mode, mode one, there is no overlap between blocks. This is the simplest mode to implement and is adequate for point operations but edge effects may cause problems if near neighbor information is used.

The three near neighbor mode provides an alternative method for near neighbor processing, especially for large window operations. The near neighbor chunks provide sufficient edge information for rotation and geometric distortion problems also.

The eight near neighbor mode, mode three, is useful when not enough near neighbor information can be obtained with mode two.

The near neighbor accessing modes (zero, two and three) must use a large part of the staging buffer to minimize the number of disk accesses. When possible, several rows of chunks will be kept there. Much use of pointers will be used in the reads and writes procedures to minimize the number of data transfers.

2.3.3 Data Reformatting

The MPP staging buffer has the capability of reformatting data flowing through it in a large number of ways. This feature has not been explicitly used in Parallel Pascal, although it is used implicitly when reading 128 x 128 chunks.

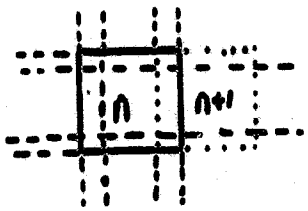
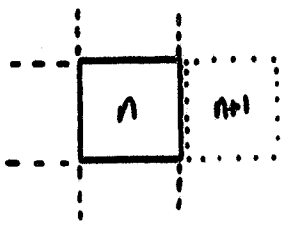
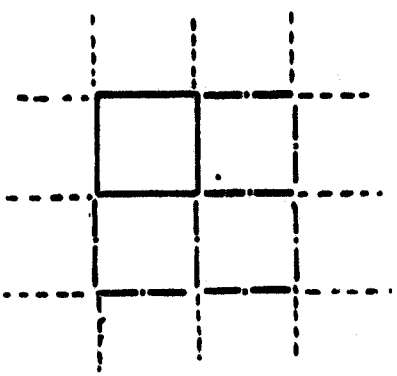
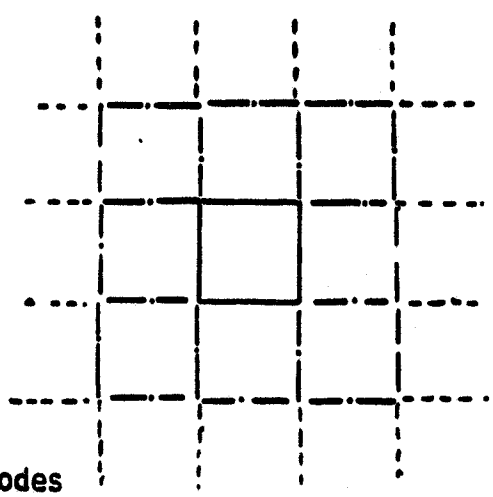
Mode	Name	Data Accessed
0	overlapped	
1	simple	
2	3 near neighbor	
3	8 near neighbor	

Fig. 2.3.1 Spooler data accessing modes

Explicit use of the reformatting feature is useful in two main applications (a) when the data file is in an unusual format and (b) when the data in the PE array is to be redistributed in the array for more efficient processing.

These functions should be treated separately, file reformatting specifies the programs view of the outside world and has no impact on the algorithm to be implemented; for this reason the format should be declared once and then conventional read and write statements may be used. On the other hand, internal data redistribution is a part of the algorithm to be implemented and this should be made explicitly clear. Methods of implementing data reformatting in Parallel Pascal are outlined below. On the MPP implementation the new functions would be built into the compiler.

File reformatting will be achieved by a procedure called reformat which takes a file identifier as an as an argument and regenerates the mapping parameters for the staging buffer. Other parameters to reformat specify the new permutation. The final parameter format for reformat has not been specified; however, a typical example using provisional format is outlined below.

Consider that a multispectral image is organized in 128 x 128 chunks of 6 bands which are interleaved and the standard Parallel Pascal I/O usually expects the data in a non-interleaved form.

The reformatting of the data can be achieved by a single call to reformat as shown in the following program segment.

```
Type
    MPPM = parallel array [ 1..6,0..127,0..127] of 0..255;
Var
    f : file of [0..127, 0..127, 1..6] of 0..255
    a : MPPM;
```

```
Begin
    reformat (f,2,3,1);
    reset (f);
    read (f,a);
      .
      .
      .
```


The file declaration specifies the external format of the data while the reformat parameters specify the dimension reordering to achieve the correct internal format.

Reformatting data in the PE array is achieved with three procedures: resetf, readf, and writef. The first parameter to resetf is a virtual channel number (an integer) which has a similar function as a file identifier except that there is no disk file associated with it. The remaining parameters of resetf completely specify the data transformation to be made.

Readf and writef are similar to read and write except that the virtual channel number replaces the file identifier. Resetf initializes the virtual channel then readf and writef can be used as if it were a conventional Pascal file. Sufficient data must be written by writef calls before readf is used.

For example, suppose that we want to combine two 128 x 128 arrays to form a 2 x 128 x 128 array, and the elements are to be shuffled together by the staging buffer. The following program segment illustrates how this might be organized

```
const
    vchan = 1;
Type
    MPPA = parallel array [0..127,0..127] of 0..255;
    MPPB = array [1..2] of MPPA;
```

```
Var
     a,b: MPPA;
     r  : MPPB;
Begin
          .
          .
          .
     resetf (vchan, <permutation parameters>);
     writef (vchan, a);
     writef (vchan, b);
     readf  (vchan, r);
          .
          .
          .
```


The virtual channel, vchan, is still open and may be used for subsequent data permutations. In general, the data transfers are not restricted to having the last dimensions 128 x 128 since the data item being transferred may have subrange indices.

For the more restrictive, but useful, case where a single array is to be permitted then a single function, called map, may be appropriate. Map is used in the following way:

```
a:= map (b, <permutation parameters>);
```

where a and b are arrays with the same number of data elements. Map may be programmed in Parallel Pascal if the functions resetf, writef and reaf are available.

3: PARALLEL PASCAL DESIGN

3.1 Motivation

Parallel Pascal is a high-level matrix language designed for the user of a parallel matrix processor. The decision to design a new language reflects a degree of dissatisfaction with facilities available in existing languages. For some reason (or combination of reasons) no single existing language was judged to be suitable for parallel matrix processors.

To judge the suitability of a language, it is necessary to consider the functions it is to serve. Wulf[1] defines three goals of a programming language: it is a design tool, a vehicle for human communication, and a vehicle for instructing a computer. The language which is chosen for a particular application should be one which satisfies all of these criteria.

Programming can be considered to be the act of mapping a problem into machine code[2]. This mapping occurs at two levels. The original problem is translated by a human into a program in some language, and then this program is translated by a compiler (or assembler) into machine code. Each translation involves the

loss of information - the program contains less information than the problem and the machine code contains less information than the program. Unfortunately, these relationships are often dual in nature - a language which facilitates programming by humans (and communication among them) will often be more difficult to compile into machine code.

In recent years, a great deal of emphasis has been placed upon the use of a set of techniques collectively referred to as ''structured programming.'' (The opinions on this subject have been by no means unanimous; a discussion of the merits and harms of a number of the ''tenets'' of structured programming is given in reference 3 among many others.) These techniques encourage careful, regular, modular designs, thereby facilitating the construction of programs which are highly reliable and maintainable.

Taking the above factors into consideration, a ''good'' language is one which facilitates communication among humans and between humans and machines, one which permits expression of a problem without undue loss of information, one which can be compiled into reasonably efficient machine code, and one which encourages structured programming techniques.

Having determined what factors are necessary for a ''good'' language, the development of Parallel Pascal can now be considered.

3.2 Parallel Pascal Specification

3.2.1 Design Goals

Since none of the available languages were entirely suitable for implementation on a parallel matrix processor, the design of a new language was undertaken. The design goal of this new language (which eventually became Parallel Pascal) for a parallel matrix processor were

- The language should be efficiently implementable. A principle reason for using a parallel processor is to obtain the maximum possible execution speed; a language whose implementation is costly significantly diminishes the advantage of parallel processors relative to more conventional (and familiar) sequential processors.

- The language must permit the direct specification of parallelism. This relates strongly to the previous objective - the direct specification of parallelism produces more efficient programs than the extraction of inherent parallelism by a compiler.

- The language must be easy to learn and use. Such a language facilitates communication among humans and between programmers and computers. A language which is difficult will be avoided by its users whenever possible.

- The language should not require the user to have an intimate understanding of the hardware upon which it is implemented.

Because Pascal is (by design) efficiently implementable and easy to learn and use, it was chosen as the basis for the new parallel matrix language. The resulting language was therefore named ''Parallel Pascal''. The following criteria were used in the specification of Parallel Pascal:

- Parallel Pascal is an extension to standard Pascal. As such, it should be fully upward-compatible; that is, any Pascal program should also be a valid Parallel Pascal program.

- Parallel Pascal extensions to Pascal should be consistent with the design philosophy of Pascal. The design should be orthogonal and the new features should not detract from the careful program construction permitted by Pascal.

When deciding upon extensions to a language, it is necessary to consider the applications for which the language will be used. Someone once said that a general-purpose system (or processor, or language) is one which does does many things but which does none of them well. To avoid the trap of implementing everything that anyone would possibly want, new features were considered in light of the desired applications area - image processing and dense matrix numerical algorithms (e.g. partial differential equations).

The following sections describe the Parallel Pascal extensions to standard Pascal. The development of each extension will be discussed.

3.2.2 Data Types

In order to satisfy the design objective that parallelism be directly expressible, a suitable data structure must be chosen. Since this specification should be as compatible as possible with standard Pascal, it is instructive to first consider the data structuring provided by Pascal. Indeed, Pascal's flexible data type facility is one of its most significant features.

The most basic Pascal data types are the predefined primitive types ''integer'', ''real'', ''char'', and ''Boolean'', and the scalar types. A user-defined scalar type associates with the type name a set of distinct identifiers. This permits the programmer to use mnemonic names rather than arbitrary integer constants, which in turn improves program readability and facilitates compile-time error checking.

The range of values which may be assigned to a scalar may be restricted by defining a subrange type. A subrange type definition comprises a base type (either a user-defined scalar type or a primitive type other than ''real'') and a range of legal values. Hence, if type ''x'' is defined by

```
type x = 1..5;
```

then a variable of type ''x'' may legally take on only the values 1, 2, 3, 4, or 5. Like simple scalar types, subrange types aid in program documentation and compile-time error checking. Also, subrange types provide information to the compiler about the amount of storage required for a variable of that type; in the

above example only 3 bits of storage need be allocated to specify any legal value in type ''x''. Providing there is hardware support, the compiler may choose to adjust the space allocated to a subrange type depending upon the available hardware representations.

A power set may be defined for a scalar or subrange type. A power set in Pascal is conceptually the same as a set in mathematics; it is a collection of elements, the composition of which changes at runtime. The base type (the subrange or scalar type over which it is defined) determines the items which may belong to the set.

There are two data structuring facilities in Pascal, the array and the record. An array is a homogenous ordered set of items. The elements of an array may be of any type: scalar, subrange, set, array, or record type. Associated with each array component is an ''index''; the range of this index may be specified by a scalar or subrange type. A record is a non-homogenous collection of items. The components of a record may be of any type, and may occur in any order. There is also a provision for the overlapping use of storage by allocating elements which are mutually exclusive into the same storage area - this is achieved through the use of a variant record.

Pascal also provides pointer types. These are defined by the compiler and initialized at runtime by the user-controlled dynamic storage allocation routine (''new''). They contain

addresses and may be copied and compared (for equality), thus permitting the construction of data structures such as linked lists, trees, etc.

The target architecture for Parallel Pascal - a parallel matrix processor - consists of a set of identical execution units which perform the same operation at the same time. The hardware thus appears as an ordered collection of homogenous processors. This organization maps naturally into the array structure provided by Pascal; hence, the array was chosen as the vehicle for the expression of parallelism in Parallel Pascal.

Often a parallel matrix processor will be closely coupled to a more conventional processor. For example, the Massively Parallel Processor contains a main control unit which is a conventional 16-bit minicomputer. In addition, the MPP is attached to a host machine, a VAX-11/780. In such an environment, it may be more efficient to perform scalar operations on one processor and matrix operations on another. This in turn is reflected in the assignment of storage to variables used in the program - those variables which are used in a scalar fashion may be physically located in a different memory than those used in an array fashion. Parallel Pascal provides for this situation by permitting an array to be declared parallel:

```
xxx: parallel array [1..5] of integer;
```

The parallel keyword is a means by which the programmer can

advise the compiler that the array (``xxx'' in this case) will be heavily used in a parallel fashion. Some compiler implementations may choose to ignore this (e.g. if there is only one type of memory). The concept that the user is advising the compiler about the implementation is similar to the register keyword in the language C[4].

It is important to note at this point that, aside from the possible difference in physical storage, arrays declared as parallel are syntactically and semantically equivalent to ``ordinary'' arrays in Parallel Pascal.

3.2.3 Array Indexing

Having chosen the array as the vehicle for expressing parallelism, it is necessary to specify the manner in which that parallelism is to be expressed. The logical starting place is the building block of any computational language - the assignment statement. It must be possible to specify the evaluation of array quantities in a simple, direct form.

Standard Pascal provides an array assignment statement; if ``a'' and ``b'' are the same type then the statement

```
a := b;
```

specifies that each element of ``b'' is to be assigned to the corresponding element of ``a''. A natural extension of this concept is to allow arrays of the same type of participate in arithmetic operations, for example, given

```
a,b: array [1..5] of integer;
i:   integer;
```

the statement

```
a := a + b;
```

would achieve the same result as

```
for i := 1 to 5 do
    a[i] := a[i] + b[i];
```

While expressions involving identical arrays are useful, they are limited in the range of problems to which they can be applied. Several deficiencies are evident. First, it is necessary to be able to select a portion of an array (for instance, a row or a column) rather than the entire array; hence, some additional indexing mechanisms are required. Second, it is necessary to allow arrays of different types (but identical shapes) to be combined in an arithmetic expression.

A number of schemes have been proposed for array indexing, as described above in the discussion of other parallel languages. The array indexing facilities which are provided must be powerful enough to solve useful problems, while remaining simple enough to efficiently implement. The choice of indexing mechanisms should therefore begin with the simplest and proceed toward the more complex.

In standard Pascal, each array index may be specified by a scalar constant or expression. This is the simplest form (and least parallel) of indexing permitted in Parallel Pascal. When

an array is indexed by a scalar its rank is (conceptually) reduced by one. When a one-dimensional array is indexed by a scalar the (logical) type of the result is a pointer to a scalar.

It was stated above that standard Pascal permits arrays participating in an assignment statement to be unsubscripted. This is actually a special case of a more general feature in standard Pascal - it is possible to elide (omit) the rightmost indices in an array assignment, provided the resulting expressions are of the same type. For example, given the definition

```
var a,b: array [1..5,1..10] of integer;
```

both of the following are legal assignments in standard Pascal:

```
a := b;
a[1] := b[1];
```

The first statement assigns to each element of ``a'' the value of the corresponding element of ``b''. The second statement performs this action only on the first row of ``a'' and ``b''. It has the same effect as:

```
for i := 1 to 10 do
    a[1,i] := b[1,i];
```

Parallel Pascal extends this to permit the omission of any index; hence, in Parallel Pascal the statement

```
a[,1] := b[,1];
```

assigns to the first column of ``a'' the values contained in the first column of ``b''. The use of a scalar index effectively

reduces the rank (number of dimensions) of an array by one; hence ''a[,1]'' is considered to be a vector.

The ability to select a row or column, as opposed to an entire array, is useful; however, it is often desirable to further restrict the number of elements which participate in an operation. A common requirement is the selection of a subset of a row, column, or both. Standard Pascal provides no symbolism to directly express this concept; hence, it is necessary to introduce a new construct to the language.

The simplest subset of a set of array indices is a consecutive range. For example, given an array with five elements, one may wish to access elements 2, 3, and 4. Standard Pascal permits the use of a subrange in type definitions to specify a range of values which a variable may possess. Parallel Pascal extends this concept by defining a subrange constant. The Pascal const statement may be used to define an identifier as a subrange constant:

```
const rangeconst = low..high;
```

A subrange constant may be added to a scalar expression and used as an array index. The most desirable syntax for this would be

```
arr[scalarexpression + low..high]
(or)
arr[scalarexpression + rangeconst]
```

where ''rangeconst'' is an identifier defined as a subrange

constant. Unfortunately, due to the recursive-descent implementation of most Pascal compilers, this syntax introduces complications when a compiler parses the program. In deference to the implementation the symbol ``@'' is used to represent the addition of a subrange constant:

```
arr[scalarindex @ low..high]
(or)
arr[scalarexpression @ rangeconst]
```

Given the following definitions:

```
const
    rr = 1..5;
    cc = 2..4;
var
    a,b: array [1..10,1..10] of integer;
    i,j: integer;
```

the following two code sequences achieve the same result:

```
(* with subrange indexing: *)
a[0@rr, 0@cc] := b[1@rr, 3@cc];

(* without subrange indexing: *)
for i := 1 to 5 do
    for j := 2 to 4 do
        a[i,j] := b[1+i, 3+j];
```

Subrange indexing does not alter the rank of an array. Thus, while ``a[1,]'' is a 10-element vector, ``a[0@1..1,]'' is a 1×10 matrix.

Other languages provide additional array indexing facilities, such as indexing by a logical set or a vector. These indexing notations are powerful, but on a processor with a limited interconnection network (e.g. a mesh network) their implementation can be very expensive. For this reason, set and

vector indexing were excluded from the specification of Parallel Pascal.

The ability to elide indices and use subrange constants for array indexing brings with it an associated problem: what combinations of array expressions are legal? In standard Pascal, the operands of an arithmetic expression must be type compatible. A subrange is type compatible with its base type, and integers are type compatible with reals. (An integer may be converted to a real number with no loss of information. Thus, if an integer expression is used where a real expression is required, e.g. on the right-hand side of an assignment statement or as an argument to a function or procedure, Pascal automatically converts the integer expression into a real expression. Since it is not true that any real may be converted to an integer with no loss of information, Pascal prohibits the opposite case - using a real expression where an integer expression is required.)

Parallel Pascal preserves the Pascal concept of type compatibility. Because of the array indexing, scalar type compatibility alone is insufficient to determine the conformability of array expressions. It is necessary to also consider the rank (number of dimensions), size, and indices of each array expression. The specification was designed to meet the following goals (note that the term ''array'' below may refer to an entire array or a subset created according to the indexing facilities described above):

- Arrays of the same type should be compatible.
- A scalar of the same base type as an array should be conformable to that array. This implies that the scalar is effectively replicated into an array of identical type.
- Arrays which differ in rank (number of dimensions) or size are not compatible. Recall that indexing with a scalar ''compresses'' a dimension out of the array.
- Arrays which have the same index ranges, but whose element types are different are compatible if the element types are compatible.
- Arrays which are the same size and shape should either be compatible, or it should be possible to make them compatible with little effort.

The first requirement above preserves the standard Pascal array assignment statement:

```
a := b;
```

where ''a'' and ''b'' are of identical types. The second requirement allows the use of scalars with array expressions; e.g. given that ''a'' and ''b'' are of the same type, and ''c'' is the same type as the elements of ''a'' and ''b'', then the following statement:

```
a := b + c;
```

adds ''c'' to each element of ''b'' and stores the result in the

corresponding element of ''a''.

Arrays are required to be the same shape and size to prevent situations such as:

```
var
    a: array [1..5] of integer;
    b: array [1..6] of integer;

b := a;
```

Allowing arrays which are the same type except for the elements, which are compatible, allows common constructions such as

```
var
    a: array [1..5] of integer;
    b: array [1..5] of real;

b := a;
```

The biggest difficulty which arises from the generalized indexing mechanisms is the compatibility of arrays whose sizes and shapes are identical, but whose index ranges are not:

```
var
    a: array [1..5] of integer;
    b: array [2..6] of integer;

a := b;
```

This problem becomes more severe when the control-flow facilities of Parallel Pascal (which are discussed later in this chapter) are used. In order to prevent ambiguity in these cases, two arrays with non-identical index ranges are compatible only if the elements of at least one are specified explictly by subrange indexing. Hence, given the ''a'' and ''b'' defined above, the

following assignments are all legal:

```
a := b[0@2..6];
a[0@1..5] := b;
a[0@1..5] := b[0@2..6];
```

The type compatiblity rules described above extend in the intuitive way to multiple dimensions. If ``c'' and ``d'' are defined by

```
var
    c: array [1..3,1..5] of integer;
    d: array [1..8,1..7] of integer;
```

then the following assignments are all legal:

```
c := d[2@2..4,0@2..6];
c[0@1..3,] := d[0@6..8,0@2..6];
c[1,] := a;
```

(Note that in the last example the ranks of ``c[1,]'' and ``a'' are the same because the scalar indexing reduced the rank of ``c'' by one.

3.2.4 Standard Functions

Pascal provides a number of standard functions and standard procedures. These perform various services, including type conversion (e.g. trunc, ord) arithmetic functions (e.g. sin, sqrt), and input/output procedures. This last group is discussed in more detail in section 3.2.6.

Many of the standard functions perform simple transformations, for instance:

```
var x, y: real;
x := sqrt(y);
```

It is quite natural to think of these functions as extensions to the set of operators provided by Pascal (e.g. ''+'', ''-''). Since Parallel Pascal allows the operators to act upon arrays as aggregates, it is only natural to extend this feature to the standard functions. Thus,

```
var
    x, y: array [1..16] of real;
    i: integer;
x := sqrt(y);
```

is effectively the same as

```
for i := 1 to 16 do
    x[i] := sqrt(y[i]);
```

These standard functions are, in a sense, ''generic'': they may be used with arrays of any shape. The value returned by the function has the same index ranges as its argument. Since these functions operate independently upon each array element, they are called elemental functions.

While the elemental functions are useful, the effective use of Parallel Pascal requires the use of functions which alter the structure of arrays in a more complex fashion. These functions r referred to as transformational functions.

The first type of array restructuring which Parallel Pascal provides is the reordering of array elements. Certain image processing algorithms (e.g. convolution) require the capability

to move data within an array. This movement may take two forms, a ''shift'', in which data is shifted off the edge of the array and zeros are brought in from the other end, or a ''rotate'', in which data is moved within an array such that data shifted off one end will reappear at the other. Parallel Pascal provides both these functions:

shift(array, i_1, i_2, i_3, ...)
rotate(array, i_1, i_2, i_3, ...)

where array is the array name and i_n is the magnitude of the shift along dimension n. (Row major order is used.)

In addition to shifting (rotating) data, it is sometimes necessary to transpose two dimensions of an array. This is performed by the trans function:

trans(array, dim_1, dim_2)

This effectively swaps two index ranges. For instance, given the definition:

```
var
    x: array [0..7, 3..4, 6..10] of integer;
    y: array [6..10, 3..4, 0..7] of integer;
    i,j,k: integer;
```

then the statement

```
y := trans(x, 1, 3);
```

is equivalent to

```
for i := 0 to 7 do
    for j := 3 to 4 do
        for k := 6 to 10 do
            y[k,j,i] := x[i,j,k];
```

The second major type of array manipulation is the alteration of the number of dimensions of an array. Some array operations may require that a array with N dimensions be combined with a array with N+1 dimensions. (One example is the computation of the matrix product $\overline{Ax}$ where $\overline{A}$ is an n×m matrix and $\overline{x}$ is an m-element vector. In this case, it would be desirable to multiply all rows of $\overline{A}$ by $\overline{x}$ simultaneously, or, equivalently, to perform an element-by-element multiplication of $\overline{A}$ and an n×m matrix $\overline{B}$, each of whose rows are the vector $\overline{x}$.) The expand function can be used to expand an array along a new dimension.

expand(array, dim, newidx)

array is either a scalar or an array. Let N be the number of dimensions of array (zero if array is a scalar). dim must be an integer constant in the range 1 to N+1. newidx is a subrange or the name of a subrange type (note: it is not a subrange constant). The array is replicated along a new dimension of type newidx which is inserted before dimension dim. For example, given the definition:

```
var
    x: array [0..7,8..15] of integer;
```

the result of

expand(x,2,5..7)

is a matrix with dimensions [0..7,5..7,8..15] in which the value of [i,j,k] is the same for all 5≤j≤7.

The last type of array operation which is frequently

required is the reduction of an array - applying an operator over a set of dimensions. For instance, one might wish to accumulate the sum along all of the rows of an array. Reduction functions have the general form:

func(array, dim_1, dim_2, dim_3,...)

where array is the array to be reduced and each dim_i is a constant expression specifying a dimension along which the reduction is to be performed. (These are required to be constants so that the compiler may determine the shape of the result.) The following reduction functions are provided:

```
sum   arithmetic sum
prod  arithmetic product
all   Boolean AND
any   Boolean OR
min   arithmetic minimum
max   arithmetic maximum
```

3.2.5 Control Flow

Standard Pascal provides several mechanisms for controlling the flow of execution. The most basic (and often overlooked) mechanism is sequencing - assignment statements are executed one at a time, in the order they appear. At a higher level, the flow of control may be altered by one of the following mechanisms:

procedures — A Pascal program consists of a set of procedures and functions (called ''subroutines'' here for convenience). A subroutine call (a procedure call or function call) diverts the flow of control to one of these subroutines. When execution of the

subroutine is complete, control returns to the statement following the subroutine call.

repetition — A statement (or a group of statements) may be executed several consecutive times by using the Pascal while, repeat-until, or for constructs. For the while construct a Boolean expression is evaluated before each iteration of the controlled statement(s); as long as this controlling expression is true the repetition continues. The repeat-until construct performs the test after each iteration rather than before it; when the termination condition is satisfied (the Boolean expression is true) the iteration stops. The for statement uses an index variable; this variable is assigned an initial value and successively incremented or decremented until it reaches a final value. The controlled statement (or statements) is executed for each value of the index.

conditional — A statement (or group of statements) may be conditionally executed by placing it in the body of an if statement. If the controlling expression evaluates to true the statements are executed; otherwise, they are skipped. Optionally, an else keyword may be specified, followed by a second statement (or block of statements); this statement

is executed if the controlling expression is false.

selection — One of a set of statements (where each ''statement'' may actually be a block of statements) may be executed according to the value of a controlling expression. This is the case statement in Pascal.

''goto'' — The flow of control may be directed to any defined statement label by use of the goto statement in Pascal. The ability to transfer control to any label within the program has been criticised as an impediment to good program design[5] It was included in Pascal because of the lack of a general agreement as to what should replace it[6] and because it is occasionally useful for breaking out of deeply-nested code structures.

All of the standard Pascal control flow constructs are present in Parallel Pascal. In order to effectively deal with arrays as aggregate entities, it is necessary to extend these constructs to deal with array operations. This extension must be carefully considered to avoid adding unnecessary complexity to the semantics of the language.

The most basic form of program construction - sequencing - is essentially the same for an SIMD-class processor (such as a parallel matrix processor) as it is for an SISD-class

(conventional scalar) processor. (This concept changes in an MIMD-class processor, since in that environment many instruction streams may be simultaneously processed.) Similarly, the concept of a procedure or function call, and the meaning of a goto are unchanged. This suggests that the extensions to Pascal will be based upon its control statements: if, case, while, repeat-until, and for.

The if statement causes the execution of one (and possibly two) statement(s) according to the value of a controlling expression. The execution is ''all-or-nothing'' - either the controlled statement is executed or it is not. This is well suited to a scalar machine, but it presents problems in Parallel Pascal. It is sometimes necessary to conditionally perform some actions using only a subset of an array. Parallel Pascal provides the where statement to address this need.

The where statement has two forms:

```
where arrayexpression do
    statement

where arrayexpression do
    statement
otherwise
    statement
```

where ''arrayexpression'' is a Boolean-array-valued expression and ''statement'' is a Parallel Pascal statement. Some restrictions apply to the controlled ''statement'':

- A goto out of the where or between the two controlled statements in a where is forbidden. (These restrictions are

imposed to facilitate the implementation of where statements with a conditional stack; uncontrolled use of the goto complicates such an implementation.)

- Array variables which appear on the left-hand side of an assignment statement must be type-compatible with the controlling array expression.

The execution of a where is defined as follows. First, the controlling expression is evaluated to obtain a Boolean array. Next, the first controlled statement (referred to later as the where clause) is evaluated. Array assignments are masked according to the Boolean array computed above. Finally, if there is a second controlled statement (an otherwise clause), it is evaluated. Array assignments within the ''otherwise clause'' are masked by the inverse of the Boolean array computed in the first step.

where statements may be nested, provided that all of the controlling array expressions are type compatible. The effect of a where statement is local to the procedure or function in which it appears; that is, it does not affect the execution of any procedures or functions called from within a ''where clause'' or ''otherwise clause''.

The where statement provides Parallel Pascal with conditional assignment (or masked assignment). That is, all array expressions within both the ''where clause'' and ''otherwise clause'' are fully evaluated, but the results are

only assigned to a subset of the array appearing on the left-hand side of an assignment statement. This allows the specification of many common problems, for instance: ''Given two arrays A and B (of the same type), determine the maximum of A and B element-by-element and store the result in A.'' This is achieved by the statement:

```
where a < b do
    a := b;
```

An alternative to conditional assignment is conditional evaluation. A conditional evaluation scheme would cause the evaluation of all array expressions to be masked (element by element) by the controlling expression. This could be used to catch exceptional conditions; for instance, divide by zero:

```
where a <> 0 do
    a := 1/a;
```

While conditional evaluation provides some additional capabilities that conditional assignment does not, it introduces semantic difficulties. One problem which conditional evaluation raises is the treatment of function (or procedure) calls from within the where statement. If an array expression is passed to a function, what values are passed for those elements for which the controlling expression is false? Similar problems arise with the use of standard functions which alter the shape of arrays - at what point is the masking applied (for at that point the expression must be type compatible with the controlling expression)? The presence of these problems with conditional

evaluation and the relative semantic simplicity of conditional assignment led to the latter's choice for the where construct.

The design of the where statement as a parallel extension of the if statement led to the consideration of a parallel extension of the Pascal case statement. The case statement selects one statement (or block of statements) from several depending upon the value of a controlling expression. It differs from the if statement in that the controlling expression is multi-valued rather than Boolean; hence, a very large number of alternatives may be selected. It was felt that a parallel version of the case statement would be used infrequently; in the interest of keeping the size of the language to a minimum it was therefore omitted from Parallel Pascal. If necessary, the effect of a parallel case statement can be achieved through the use of a series of where statements (in the same fashion as a standard Pascal case statement can be implemented by a series of if statements).

The only remaining control constructs to be considered are the loop structures while, repeat-until, and for. The loop is one of the biggest sources of error for programmers; therefore, adding complexity to the looping mechanisms seemed unwise. It is unclear how a for loop should be extended. Further, a combination of a standard Pascal while or repeat-until loop statement (perhaps using a reduction function such as ''any'' or ''all'' to use conditionals based upon entire arrays) and a where statement can express all of the operations that any new loop construct of moderate complexity could express.

3.2.6 Input and Output

Pascal provides a fairly minimal set of input and output procedures. Each file consists of a uniform sequence of objects of a fixed type. The file is accessed by means of a ''buffer variable''. Syntactically, file buffer variables are used in the same fashion as pointers. To perform output, the data is placed into the file buffer and the ''put'' procedure is called. To perform input, the file variable is read, after which the ''get'' procedure is called to advance to the next item in the file. The ''eof'' function may be used to determine whether a file is positioned at the end. The ''reset'' procedure repositions a file at the beginning and makes it available for reading, while the ''rewrite'' procedure repositions a file at the beginning after truncating it, and makes it available for writing.

In addition to the ''get'' and ''put'' procedures, the ''read'' and ''write'' procedures may be used. Given the file ''f'' and variable ''x'' (both having the same type) the following equivalences hold:

```
read(f,x)    ≡    x := f↑; get(f)
write(f,x)   ≡    f↑ := x; put(f)
```

Files whose elements are of type ''char'' (i.e. those of type ''text'') are treated specially. The procedures ''read'' and ''write'' may be used to transfer numeric data to or from a text file - the appropriate conversion is performed. In addition, the procedures ''readln'' and ''writeln'', and the

function ''eoln'' are provided for intelligent handling of line-formatted input.

The use of text files for mass information input and output on a parallel processor was considered highly unlikely. Therefore, it was decided that Parallel Pascal needed no additional provisions for dealing with text files beyond those provided by standard Pascal.

On the other hand, it was apparent that ''binary format'' input and output would be heavily used. In particular, the limited main memory of a matrix processor implies that a great deal of data movement will be performed during the execution of a program. This subject falls in a ''gray area'' between the specification of the language and its implementation, for the manner in which the main memory of the parallel processor is managed directly affects the type of input and output required. For these reasons, it was decided to retain standard Pascal input and output without extensions for the definition of Parallel Pascal. The facilities which are required for memory management can best be determined after a period of use. Additional standard functions (which can be added to the language without significant trauma) could be added at a later time if a definite need arose. (Another, less desirable possibility, would be the inclusion of some standard procedures on a site-dependent basis. This is in fact likely in other areas of Parallel Pascal, e.g. the implementation of interconnection functions which are more complicated that the simple mesh network defined for Parallel

Pascal, but which are supported by a particular machine.)

A discussion of some of the possible input and output facilities for managing a limited memory is presented in Section 5.

3.3 References

1 William A. Wulf, "Trends in the Design and Implementation of Programming Languages," IEEE Computer, pp.14-22 (January 1980).

2 Leslie Lamport, "On Programming Parallel Computers," ACM SIGPLAN Notices Vol. 10(3), pp.25-33 (March 1975).

3 Paul Abrahams, "Structured Programming Considered Harmful," ACM SIGPLAN Notices Vol. 10(4), pp.13-24 (April 1975).

4 Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Prentice-Hall, Englewood Cliffs, NJ (1978).

5 Edsger W. Dijkstra, "Go To Statement Considered Harmful," Communications of the ACM Vol. 11(3), pp.147-148 (March 1968).

6 Niklaus Wirth, "An Assessment of the Programming Language Pascal," IEEE Transactions on Software Engineering Vol. SE-1(2), pp.192-198 (June 1975).

4: PARALLEL P-CODE

In this section the development of Parallel P-code is described. Appendix B contains a complete description of Parallel P-code.

4.1 Pseudo-code

The concept of pseudo-code (``P-code'') was introduced by Urs Ammann with the portable Pascal P4 compiler[1]. P-code is a simple, fixed format language representing the assembly language of a hypothetical stack computer (``P-machine''). The low-level, implementation-dependent details (e.g. the internal representation of the various data types) are not specified. The operators in P-code were chosen to closely reflect the architecture of contemporary computer systems; hence, code generators (to convert P-code to native machine code) can be constructed fairly easily. Alternately, the P-code can be converted from its symbolic form to a binary form and executed by an interpreter.

The structure of P-code reflects the structure of Pascal. The P-machine upon which P-code runs is a stack-oriented computer. All procedure activation records are maintained on the

stack, permitting access to variables according to their lexical level (static nesting) and offset within the procedure activation. There are no directly-accessible registers. Instructions and data are completely separate, and all data memory locations (whether on the stack or the heap) are strongly typed. The P-code instruction set includes instructions to check array bounds.

Since P-code was introduced, some variants have been developed. Reference 2 compares standard P-code (also called P-4 P-code) with variants developed at the University of California at San Diego and at Los Alamos Scientific Laboratories. These variants were motivated by implementation needs. In the case of UCSD, a efficient and compact form was needed for execution on micro- and mini-computers with a limited address space. In the case of LASL, extensions were needed to fully utilize the target machine (a CRAY-1) and to interface with Fortran programs.

An alternate intermediate language form which was also considered was a tree-structured language. One example of this form is ``T-code'', a language defined by the Systems Research Group of the University of Illinois[3] for use in another compiler. T-code is a directed acyclic graph based upon Pascal expression trees. The representation of Pascal expressions as trees, rather than as a linear sequence of P-code instructions, can facilitate optimization and code generation.

Since the P4 compiler was selected as the basis for

implementing a Parallel Pascal compiler, it was decided to define an intermediate language based upon P-4 P-code. While a compiler which directly produced an intermediate language such as T-code was considered, the simplicity of a P-code-like language, the P-code nature of the P4 compiler, and the fact that the Illinois T-code generator accepted P-code as input swayed the decision in favor of the more conventional pseudo-code format.

Although a P-code format was chosen as the intermediate language, standard P-code was inadequate to represent the data structuring and aggregate operations that are required for a parallel matrix processor. This led to the development of a new intermediate language, based upon conventional P-code, called ''Parallel P-code''. The following sections describe the development of Parallel P-code.

4.2 Data Types

The most significant difference between standard P-code and Parallel P-code is the way in which they treat data types. In standard P-code, only a few data types are supported - integer, real, Boolean, character, set, and pointer. These are sufficient to perform all operations in standard Pascal, because Pascal deals with data on an element-by-element basis. Parallel Pascal, however, permits (and in fact encourages) the manipulation of arrays as aggregates. The problem of array aggregate operations

can be solved in one of two ways.

The first alternative is to define a small number of new data types representing the array types that the matrix processor can act upon directly. A set of fundamental operations would be defined upon these basic data types. Larger operations would then be explictly ''unrolled'' by the compiler into a sequence of these fundamental operations. This approach positions the intermediate language at a low level; the pseudo-code is nearly the assembly language of the target machine with the syntax ''cleaned up''.

The second alternative is to treat all operations at a high level. Rather than having a finite set of fundamental types, the compiler would define new data types and would specify array operations with a single instruction instead of an unrolled sequence. This approach positions the intermediate language at a much higher level than the first approach.

Of the two schemes, the first method requires more work by the compiler ''front end'' and very little work by the ''back end'' (code generator). The second method requires a great deal more of the code generator. However, the intermediate language for the second method is much more machine-independent, and with its higher information content it facilitates optimization. Parallel P-code is designed according to the second approach.

The base types defined in Parallel P-code are very similar to those in standard P-code: integer, real, character, and

Boolean. With the appropriate definition statements, these base types are used to define all structured types.

In the following discussion, the definition statements are referred to as ''pseudo-operators'' or ''pseudo-ops'' since their role in Parallel P-code is very similar to the role of pseudo-operators in a conventional assembly language.

4.2.1 Subrange Types

Standard P-code uses objects of type ''integer'' to hold values of a subrange type. While this is suitable for a conventional word-oriented machine, a bit-addressable machine (such as a bit-serial matrix processor) can utilize memory more efficiently by only allocating the minimum number of bits needed to represent all values within the subrange. Parallel P-code provides the .RANGE pseudo-op to declare a subrange; for example, the type ''rng'' can be defined to be the integers from 1 to 5 with the statement:

```
.RANGE    rng,1,5
```

The base type for a subrange is always ''integer''. As in standard P-code, integers are used to represent user-defined scalar types. There is no provision for a subrange of characters - the standard character type is used instead.

4.2.2 Set Types

In standard P-code there is only one type for sets. The P4 compiler implementation notes[1] recommend the use of a bitstring to implement a set. Limiting the set to one representation restricts its generality in two ways. First, the maximum number of elements in the set is fixed. Second, the range of the elements themselves is restricted. That is, if there are numset possible elements, then they are represented by the integers 0, 1, ... numset-1. Integers which fall outside of this range cannot belong to a set.

Parallel P-code permits the definition of a powerset type with the .SET pseudo-op. For example, the type ``pset'' can be defined to contain the integers from 5 to 10 with the statement:

```
.SET pset,5,10
```

The base type for a powerset is always ``integer''. As in standard P-code, integers are used to represent user-defined scalar types.

Parallel P-code does not define the format that a powerset is to have; instead, it is left to the implementation. However, it is occasionally necessary to specify a powerset constant. The constant is specified by the type of the set and the elements; e.g. the powerset constant ``[5,6,9]'' of type ``pset'' would be represented in Parallel P-code as

```
pset,(5,6,9)
```

It is necessary to specify the type name as well as the members, because different sets may have different implementations.

4.2.3 Files

The P4 compiler only permits files to be of type ''text'', that is, ''file of char''. Thus, there is no need to distinguish files of different types in standard P-code. Parallel P-code provides the .FILE pseudo-operator for specifying the type of a file. The syntax is intuitive; to define ''ftype'' as a file with elements of type ''etype'' the statement is:

```
.FILE     ftype,etype
```

4.2.4 Array Types

In standard P-code, almost all operations are performed on scalar elements. (The exception to this rule is a provision for moving blocks of data from one place to another.) Parallel Pascal, however, requires operations to be performed upon arrays as aggregates. As discussed above, the decision was made to provide a formalism for specifying these parallel operations in the intermediate language.

In order to process array operations, the code generator must know at least the size of the array and the type of elements. For more sophisticated operations (e.g. operations involving only a subset of the array) it must also know the layout of the array - the number and range of array dimensions. This information can be divided into two portions, static and

dynamic.

The static portion represents information that is known at compile time. It consists of such things as the base type (i.e. the type of the array elements), the number of dimensions, and the low and high bounds of each dimension. This portion can be considered the logical specification of the data.

The dynamic portion of an array type consists of the address of the array and the specification of which elements are to participate in an operation. This portion therefore represents the physical specification of the data - where it is stored and what portions of it (e.g. which array elements) are to be affected.

The static and dynamic information is collectively referred to as an array descriptor. The parallel languages ALA[4] and LRLTRAN[5] also contain array descriptors, but there are several significant differences between those descriptors and Parallel P-code descriptors. The descriptors in ALA and LRLTRAN are user-accessible, while the descriptors defined here are not directly user-accessible. Parallel Pascal contains no concept of an array descriptor; they are defined only in the Parallel P-code implementation. The size of the data referenced by an ALA or LRLTRAN descriptor may be varied by the user; Parallel Pascal descriptors by contrast always refer to data whose size is fixed. (Both types of descriptors allow selection of a subset of the

elements to which they refer.)

The static portion of an array descriptor is specified in Parallel P-code via the .ARRAY pseudo-operator. The base type (i.e. array element type), number of dimensions, and range of all dimensions are specified. For instance, the array type defined by

```
arr = array [1..5,2..6] of integer;
```

would be defined in Parallel P-code with the statement:

```
.ARRAY    arr,integer,2,1,5,2,6
```

An array is never defined in terms of another array; thus, the following definitions:

```
row = array [1..5] of real;
mat = array [4..8] of row;
```

will be translated to Parallel P-code as:

```
.ARRAY    row,real,1,1,5
.ARRAY    mat,real,2,4,8,1,5
```

Parallel Pascal provides the parallel reserved word for declaring that an array should be allocated in the parallel array memory rather than the sequential control unit memory. If an array is declared parallel, this fact is reflected in Parallel P-code by a negative rank. For instance,

```
arr = parallel array [2..4,8..16] of integer;
```

is translated to

```
.ARRAY     arr,integer,-2,2,4,8,16
```

The dynamic portion of array descriptors will be dealt with in more detail in a later section.

4.2.5 Record Types

In order for arrays of records to be intelligently processed, it is necessary for the intermediate language to define descriptors for records, as well as arrays. Like array descriptors, record descriptors consist of a static and a dynamic portion. The static portion specifies the record: the fields and their types. The dynamic portion specifies the address of the record and the field which has been selected for a particular operation. (Unlike arrays, it is not possible that more than one field in a particular record will be simultaneously selected. This property is a result of the choice of the array, rather than the record, as the data structure used to express parallelism, as described in chapter 2.)

Because the structure of a record is not as regular as the structure of an array, a single type definition statement for the static portion of a record would be cumbersome. For that reason, Parallel P-code defines records according to the fields which they contain. The pseudo-operator used to define record components is .RECORD. One .RECORD is generated for each field.

Parallel Pascal, like standard Pascal, permits variant records. When a record has variants, several components will

share the same memory allocation. (Only one is in use at any given time.) Parallel P-code permits the specification of an offset with each field declaration. A record definition in Parallel P-code consists of a sequence of .RECORD statements. Normally, each successive field in the same record is assigned a sequential location in memory. However, this behavior can be overridden so that a field is aligned at the same offset as a previous field.

The general syntax of the .RECORD pseudo-op is

```
.RECORD   rname,fname,offset,ftype
```

where ``rname'' is the name of the record being defined, ``fname'' is the name of the field being defined, ``ftype'' is the type of the field, and ``offset'' is either ``nil'' or the name of a previously-defined offset. If ``offset'' is the literal string ``nil'', the next sequential memory location is assigned; otherwise, the new field ``fname'' is aligned with the existing field ``offset''. As an example, the record defined by:

```
rec = record
    x: integer;
    y: real;
    case Boolean of
     false: zf: integer;
     true:  zt: real;
    end;
```

would be translated to

```
.RECORD   rec,nil,x,integer
.RECORD   rec,nil,y,real
.RECORD   rec,nil,zf,integer
.RECORD   rec,zf,zt,real
```

4.2.6 The Dynamic Portion of Descriptors

In order to examine the specification of the dynamic portion of array and record descriptors, it is necessary to first consider the way in which they are to be used.

As discussed above, standard P-code is the assembly language of a hypothetical stack computer. Parallel P-code was also designed with this general philosophy. All operations are performed by means of a run-time stack. Data is loaded onto the top of the stack, manipulated on the stack, and stored from the top of the stack. In standard P-code, data is manipulated in one of two ways. The first way is to load the data onto the stack and manipulate it directly. This is the most common method (in standard P-code) and it works well because Pascal usually deals only with one item at a time. An alternate way is to perform a data transfer of a compile-time specified number of elements between two addresses which are computed at runtime. In this second case (used in assignment statements where both sides are identical arrays or records), the addresses, not the data, reside on the stack. They could be called very simple descriptors because they describe where the referenced data is (or is to go).

It seems reasonable that Parallel P-code should also make use of these two mechanisms. When an operation is performed on scalar data, the data itself is loaded onto the runtime stack, manipulated, and stored from the stack. When an operation involves an array or record, or some combination thereof, the

second method is called for. However, because Parallel Pascal provides more flexibility in aggregate operations, an address alone is not sufficient. Unlike the standard P-code case, which involved a typeless move of a consecutive block of data from one address to another, information must be provided about the shape and type of the data. The type information is supplied by the static descriptor (i.e. by an .ARRAY or .RECORD pseudo-operator). The runtime-dependent shape information is provided by the dynamic descriptors on the runtime stack.

Dynamic descriptors on the runtime stack in Parallel P-code are most easily understood when considered recursively. Each level of structuring is applied to a descriptor formed at a higher level. Before exploring this concept completely, an examination of the format for array and record descriptors is in order.

The runtime nature of an array is determined by two dynamic attributes: the address of the array and the index ranges of its dimensions. The dynamic (physical) portion of the array descriptor which resides upon the runtime stack specifies these attributes. This information is constructed by loading a ''blank'' descriptor (one which specifies the array address but does not specify index ranges) and then ''filling in'' the index ranges using one of three operators: IX0 (select entire index range), IX1 (index by a scalar), or IX2 (index by a subrange). Each successive index instruction is applied to the next unspecified array index range. Note that the compiler does not

know or care what in what format the dynamic array descriptor is specified.

The concept that indexing by a scalar is to reduce the rank of the array (e.g. a column of a matrix is considered to be a vector) requires extra attention. The static type of the top-of-stack is changed by scalar indexing. This represents the logical type of the data. Parallel P-code does not specify the impact upon the dynamic portion of the descriptor, which indicates the physical attributes of the object. In the hypothetical machine which implements Parallel P-code, the dynamic descriptor still specifies the physical memory associated with the array, even though the type of the array has changed. A code generator (which does not actually simulate a runtime stack) must similarly ''remember'' the physical origins of an array whose logical shape has been altered by scalar indexing.

In contrast with arrays, only one component of a record may be specified at a time. However, unlike arrays, the fields in a record are non-homogenous. The manner in which the target machine stores the fields of the records will affect how a record field is specified; the compiler cannot simply calculate a constant offset (as is done in standard P-code). Word sizes differ between machines - one machine may store both integers and floating-point numbers in the same size word, while another may require several units of storage for a floating-point number. A further complication is introduced by the architecture of the intended target machine (a parallel matrix processor), because it

will usually contain two non-identical memories for scalar and array data. One of the design goals of the intermediate language was to be relatively implementation-independent. In addition, there was a strong desire to keep Parallel P-code at a high level of abstraction to simplify the ''front end'' and retain as much symbolic information as possible for the ''back end'' to use for optimization and code generation. Therefore, all record offsets in Parallel P-code are made by means of symbolic names. The names correspond to the field names defined in .RECORD statements.

The exact format of a record descriptor is not known to the ''front end''. Instead, the record descriptor is constructed with the aid of the ''select'' (SEL) instruction. A descriptor that specifies the entire record is loaded onto the stack; this is similar to the ''blank'' descriptor described above for arrays but may be used without further modification to access the entire record. The SEL operator is used to select a field from the record. This replaces the record descriptor on top of the stack with a modified descriptor that indicates the address of the record and the selected field. If that field is itself a record, another SEL is then used to select a field within that sub-record.

The SEL operator, like the IXI operator, changes the logical type of its operand from a record to a record field. As with the array case, the dynamic descriptor will still contain information

about the physical storage associated with the new logical type.

Descriptors for more complex structures (e.g. arrays of records, arrays within records) are constructed by repeated application of the techniques above. For instance, given the following:

```
arrec:  array [1..5] of
        record
            x: array [1..10] of integer;
            y: integer;
        end;
```

a descriptor for ``arrec[@1..2].x[i@0..2]'' would be constructed by the following steps:

1. Load a ``blank descriptor'' (which specifies the address but no index ranges) of ``arrec'' onto the runtime stack.

2. Load the constant 0 onto the stack. Perform an IX2 operation using the subrange ``1..2''. The stack now contains a descriptor for an array of records.

3. Perform a SEL to select the field ``x'' in the records described by the descriptor on the stack. The stack now contains a descriptor for a two-dimensional array, for which the first index range has been selected as ``1..2'' and the second is (as yet) unspecified.

4. Load the value of ``i'' onto the stack. Perform an IX2 operation using the subrange ``0..2''. The stack now contains a descriptor for a 2×3 array whose dimensions have been selected as ``1..2'' and ``i..(i+2)''.

4.2.7 Pointers

In standard Pascal, a pointer is simply the address of a data item. The pointer can be copied and compared, but its value cannot otherwise be affected by the programmer. Parallel Pascal provides the same symbolism for specifying pointers that standard Pascal does. However, the implementation of a pointer as simply an address limits its usefulness in Parallel P-code.

As the previous section discussed, the dynamic portion of an array descriptor, a record descriptor, or a hybrid of both, consists of an address and information about which dimensions (or fields) are selected. Once this information has been constructed on the runtime stack, it can be used as an address for P-code operations (for example, loads and stores). Normally, the descriptor is used in the process of manipulating the data it describes, but at times it is necessary for the descriptor itself to be manipulated. (These operations are compiler-generated, since Parallel Pascal does not provide the concept of a descriptor.) For this reason, Parallel P-code implements all pointers as descriptors. Descriptors of scalar data are simply addresses; hence, for scalars the concept of a pointer is unchanged.

Descriptors (pointers) are defined in Parallel P-code with the ``.POINT'' pseudo-operation. For instance, to define type ``abc'' as a pointer to type ``xyz'' the statement would be:

```
.POINT    abc,xyz
```

4.2.8 Type Renaming

Occasionally, to expedite processing by the compiler front-end, it is convenient to refer to two identical types by different names. Parallel P-code provides the ''.TYPE'' pseudo-operation for this purpose. The statement

```
.TYPE     xxx,yyy
```

defines type ''xxx'' to be the same thing as the already-defined type ''yyy''.

4.3 Memory Allocation

There are two types of variables in a Pascal program - those which are allocated on the runtime stack and those which are allocated dynamically from a runtime heap. The former correspond to the ordinary variables declared by a subroutine (function or procedure) - they are automatically created upon subroutine entry and automatically deleted upon its exit. The latter correspond to the pointer variables - the pointers themselves are allocated upon entry to a subroutine but they reference memory which is allocated by the procedure ''new'' and released by the procedure ''dispose''. Like Pascal, Parallel Pascal uses both types of memory allocation.

In standard P-code the local variables for each subroutine are allocated on the runtime stack by reserving a consecutive block of stack memory. A special instruction, ENT, specifies the

number of arguments to the subroutine and the number of memory units required for local variables and temporary storage. The compiler which produces the P-code ''knows'' the memory requirements of each type of variable; thus, it can calculate the offset within this consecutive block of each local variable contained therein. In the case of an array, the elements of the array are stored consecutively in row-major order; the compiler can compute the address of any element according to the usual formula.

A typical Pascal program will contain a main program and one or more user-defined functions or procedures. Because Pascal is a block-structured language, procedure and function definitions are nested; that is, the definitions of some subroutines will be contained within the main program, and some of these subroutines will themselves contain the definition of other subroutines. This is referred to as the static nesting of the program. Each procedure is associated with a lexical level. The outermost block contains the main program and the global variables; these are located at lexical level 0. If a function or procedure definition is contained inside a block at level i, then that function or procedure is at level i+1. Functions and procedures at level i can reference all of the variables and invoke all of the procedures and functions defined in the i-1 containing blocks.

Pascal permits recursive function and procedure calls. Each time a function or procedure at level i is called a new set of

local variables is allocated. Thus, when a function or procedure at level i accesses a variable at level j it is accessing the variable corresponding to the most recent set of allocations at level j. Unlike the static nesting, the sequence of memory allocations, called the dynamic chain, will vary at runtime.

Corresponding with each called function or procedure is an area on the runtime stack called the stack frame (or activation record). In addition to the arguments to the function or procedure, the local variables, and space for temporary results, the stack frame includes some linkage information. In standard P-code this includes the return address, space for a returned function result (this field is unused for procedures), and two locations for the static and dynamic links. The static and dynamic links point to the appropriate previous stack frames. The hypothetical machine which implements P-code contains a non-user-accessible register called the ''frame pointer'' which holds the address of the current stack frame.

Because of the dynamic nature of the memory allocation, it is not possible to compute the absolute addresses of any data (except for variables in the outermost - global - block). Instead, the desired locations are obtained by using a two-level lexical-level addressing scheme. The form of a lexical-level address is

(level, offset)

where ''level'' is the static nesting level and ''offset'' is the

offset of the variable relative to the beginning of the stack frame which contains it. Standard P-code uses a modified version of this scheme. Rather than specifying the lexical level directly, it instead specifies the difference between the current lexical level and the lexical level of the desired operand. Thus, if the current lexical level is 4 and the desired variable is at offset 43 at lexical level 1, the lexical address is (1,43), which standard P-code expresses as (4-1,43) or (3,43).

The use of lexical-level addressing is a powerful technique. However, the allocation of memory directly on the runtime stack presents problems for Parallel P-code. First, one of the goals is that Parallel P-code be machine-independent. This precludes the use of compiler-calculated offsets for variables within a stack frame, since word sizes and data representations vary from machine to machine. Second, a parallel matrix processor which contains more than one type of memory (e.g. array memory and scalar memory) cannot allocate all variables on one stack. Therefore, Parallel P-code implements a modified form of lexical addressing.

Parallel P-code represents lexical addresses directly, rather than subtracting the lexical level of the operand from the current lexical level. This definition is more intuitive and constitutes no loss of information. Parallel P-code does not define the exact format of a stack frame; specifically, it does not define the format of the static and dynamic links. These are left to the implementation. This provides a degree of

flexibility - an implementor may wish to use a display[6] rather than an explicit static link chain.

To eliminate the need for compiler-generated offsets in a lexical address, Parallel P-code uses a symbolic form of lexical addressing. Each function or procedure argument and each local variable is assigned an index number. The lexical address consists of the lexical level and the index number. For instance, given the function:

```
function func(a,b: real) : integer;
var x,y: integer;
```

the function result is index 0, ''a'' is index 1, ''b'' is index 2, ''x'' is index 3, and ''y'' is index 4. If this function is at lexical level 5 the lexical address of ''x'' would be (5,3).

Local variables, arguments to the function or procedure, and the result (if the routine is a function) are specified in Parallel P-code with the .ARG and .LOCAL pseudo operators. Arguments and local variables share the same set of indices; however, arguments require special treatment and therefore are warranted a separate declaration statement. The index 0 is reserved for the result of a function. It is unused for procedures. Arguments are defined with the syntax:

```
.ARG index,type,rv
```

where ''index'' is the index number, ''type'' is a type name, and ''rv'' is zero if the argument was passed by value or one if it was passed by reference. Local variables are declared with a

similar statement:

```
.LOCAL    index,type,overlay
```

The ''index'' and ''type'' fields are identical to those for .ARG. The ''overlay'' field is similar to the ''align'' field for the .RECORD pseudo operator. It is normally zero, indicating that the local variable should be allocated the next available memory location (or locations). If it is non-zero, it specifies a previously-defined local variable (at the same lexical level); the new variable is to be overlayed on the memory allocated for the specified old variable. The use of this feature to implement the with statement is described below.

Parallel P-code also defines explicitly the lexical level or each procedure of function. Each routine is preceeded by an .ENTRY statement and followed by an .EXIT statement. These specify the lexical level of the enclosed procedure. The .ENTRY statement also specifies the processor on which the procedure or function is to run:

```
.ENTRY    level,site
.EXIT     level
```

''level'' is a lexical level number, and ''site'' is either the literal string ''HOST'' indicating that the routine is to be executed on the host machine, or ''MCU'' indicating that it is to be executed on the (main) control unit of the parallel processor.

There are no Parallel P-code instructions which directly allocate or release dynamic memory. These operations are

performed by the standard procedures ''new'' and ''dispose''. These procedures operate in Parallel P-code the same way as in standard P-code, except that they may return either a scalar memory pointer or an array memory pointer. They accept as an operand a pointer variable.

4.4 Data Manipulation

4.4.1 Overall Strategy

As the previous sections have discussed, Parallel P-code, like standard P-code, is a stack-oriented language. This section describes the overall data manipulation scheme in Parallel P-code. The specific opcodes provided in Parallel P-code are described in full in Appendix B.

Conceptually, the runtime stack for Parallel P-code contains quantities which are either scalars or are descriptors for an array or record type. At times, the stack also contains pointers to scalars (one might consider these to be scalar descriptors).

When an operation is performed on scalars, the address where the result is to be stored is loaded onto the stack, the scalar expression is calculated, and a ''store indirect'' is performed to store the result of the expression (on top of the runtime stack) at the specified address (the second item on the runtime

stack).

In order to generalize the scalar case to structured types (arrays and records), it is necessary to define what is meant by a ''load'' of an array or record. In Parallel P-code these are always accessed through a descriptor. Since a descriptor is essentially a generalized pointer, there is a fundamental difference between manipulating scalars and manipulating structured types. In the former case, the value of the scalar is loaded onto the stack, manipulated, and stored. In the latter case there is an additional level of indirection.

When an operation is performed, the result must be stored in a temporary area and a descriptor for that temporary area placed upon the runtime stack. Considered in this fashion, the descriptors for the defined local variables are analogous to the addresses of scalar variables, and the descriptors of temporaries are analogous to scalar values on the stack. The automatic allocation of the temporary storage to which the descriptors refer is the responsibility of the implementation.

4.4.2 Load Instructions

Parallel P-code provides five instructions for loading data onto the runtime stack.

The simplest instruction is LDC, which loads a constant. The constant is never an array or record. The constant may be an integer, floating-point number, character, Boolean value, or set.

The specified constant is pushed onto the top of the runtime stack.

Two instructions are provided for loading addresses. The first, LCA, is used to load the address of a constant. The constant is specified as in the LDC instruction. Rather than loading the value of the constant, however, LCA creates a constant in memory and loads its address onto the stack. This instruction is used when it is necessary to pass a constant string ''by reference'' to a procedure or function. The second instruction, LLA, converts a lexical address (level,index) to an absolute address and pushes the address on the runtime stack. If the item is an array an array descriptor which specifies the location of the array but no indexing information is pushed; similarly, if the item is a record a descriptor which specifies the entire record will be pushed. (Hybrids of arrays and records are handled in the same fashion, as discussed in section 4.2.6.

Data is loaded by means of the LOD and LDI instructions. LOD is used to load scalar values whose (lexical) address is known at compile-time. LDI deserves detailed attention.

The syntax for LDI is:

LDI type

where ''type'' is the type of the data to be loaded. The top of the runtime stack contains a descriptor for the data to be loaded. If ''type'' is a non-array, non-record type, this descriptor is simply an address. In this case, the specified

data is loaded onto the stack. If ''type'' is an array or record type, the addressed data is copied to a temporary location (in either the array memory or the scalar memory) and the top of the runtime stack is replaced with a descriptor to the temporary location.

Usually, an LDI of an array or a record is redundant because the descriptor will only be used as the input to a subsequent operation (e.g. ADD). However, in some cases it is necessary to preserve the distinction between the variable itself and its value. For instance, if a variable is passed ''by value'' to a function, it is not acceptable to pass the original array descriptor; instead, a descriptor for a copy of the array must be passed. Where this distinction is not necessary, the implementation may choose to ignore the LDI (e.g. a simple optimization would be to omit any LDI whose result is used in a subsequent expression).

When an LDI is performed on a file, the file buffer variable is loaded onto the stack.

4.4.3 Store Instructions

Parallel Pascal provides only two store instructions, STO and STO. STO is used to store non-array, non-record data at a (lexical) address which is known at compile-time. STO is used to store all forms of data (on top of the runtime stack) into the variable specified by the descriptor which is next to the top of stack. In the case of a non-array, non-record, this descriptor

is simply an address. In this case, the top of stack is copied into that address. Otherwise, the data indicated by the descriptor on top of the stack is copied into the area indicated by the descriptor next to the top of the stack.

When an STO is performed using a file as the address, the top of stack is stored in the file buffer variable for the specified file.

4.4.4 Type Conversions

There are two mechanisms by which the type of an item on the runtime stack may be altered. It may be explicitly coerced by the CVT or CVN operator, or, if it is a record type, a field may be selected with the SEL operator.

The CVT and CVN operators are used to perform a variety of type conversions. They are essentially the same operator, except that CVT operates upon the top of the runtime stack, while CVN operates upon the next-to-top of the runtime stack. The syntax for these operators is

```
CVT  oldtype,newtype
CVN  oldtype,newtype,tstype
```

where ''oldtype'' is the old type of the item to be converted, ''newtype'' is the type it is to be converted to, and ''tstype'' (for CVN) is the type of the top of stack (this information is needed because stack items may be different sizes).

CVT and CVN perform three major functions. First, they

convert scalars or arrays of a simple type from one base type to another. An example of this is the conversion of an array of integers to an array (with the same shape and array indices) of real numbers. Second, they collapse one-element arrays into scalars. For this case, the array descriptor specifies a single element. Third, they expand scalars into arrays. In this case, every element of the resulting array has the value of the scalar. The role that these scalar-to-array conversions play is discussed in more detail below.

SEL is used to select a field from a record. The syntax is:

SEL rectype,field,newtype

where ''rectype'' is the type of the record, ''field'' is the name of the field to be selected, and ''newtype'' is the type of the result. ''newtype'' is not necessarily the type of ''field'' - if ''rectype'' is an array of records then ''newtype'' will be an array also. The record descriptor on top of the runtime stack is modified to include the additional field selection information.

4.4.5 Conformability

Parallel Pascal has strict rules regarding the conformability of two items which are used together. The conformability rules ensure that the specified operation is well-defined and efficiently implementable. Operands in Parallel P-code are also required to be conformable, although the requirements are less rigid than those in Parallel Pascal. An

operation which is performed on two non-conformable items is an error. The disposition of this error condition is left to the implementation.

In Parallel P-code, the operands of a binary operation (this class includes the ''store'' instructions) must be conformable in two ways. First, the base types of the operands must be identical. For instance, it is illegal to combine an integer (or an array of integers) with a real number (or an array of real numbers) without first explicitly converting one of the operands so that both are integers or both are real numbers. This conversion is performed by the CVT and CVN operators. It is also illegal to combine an integer and a value of subrange type without first explictly converting one operand so that the types match.

In Parallel Pascal, arrays may be combined with scalars, and two arrays of the same shape may be used together. (More precisely, two arrays whose non-scalar index ranges are identical or explictly specified, and which have the same shape may be used together.) In Parallel P-code, the operands to an instruction must always be the same type. If a scalar is to be combined with an array, the scalar must first be expanded to an array of the same shape. This expansion creates an array descriptor with ''blank'' indexing information. For instance, if the top of the runtime stack contains a descriptor for the array defined by

```
var
    a: array [1..5] of integer;
```

This might be defined in Parallel P-code as:

```
.ARRAY    T8,integer,1,1,5
.LOCAL    1,T8,0
```

To increment all elements of ''a'' by one, the following sequence could be used:

```
LLA  0,1  ;array descriptor for ''a''
IX0  T8
LLA  0,1  ;array descriptor for ''a''
IX0  T8
LDI  T8   ;load ''a''
LDC  integer,1
CVT  integer,T8     ;convert scalar to array
IX0  T8   ;define index range for new array
ADD  T8
STO  T8
```

The LDC places the integer constant 1 onto the top of the stack. The CVT expands the top of the stack into an array of type ''arr'', every element of which contains the value 1. (The resulting array is allocated in temporary memory and its descriptor replaces the integer on top of the stack.) The top of stack is an array descriptor with ''blank'' indexing information, so the IX0 is used to select every element of the (newly-created) array. The ADD then adds together the two arrays whose descriptors are on top of the stack.

The creation of a ''blank'' array descriptor by the CVT instruction allows a scalar to interact with any subset of an array. In the previous example the entire array was selected; however, a subset can also be easily incremented. The operation

```
a[0@1..2] := a[0@1..2] + 1;
```

would be implemented by

```
LLA   0,1
LDC   integer,1
LDC   integer,2
IX2   T8    ;array descriptor for ''a[@1..2]''
LLA   0,1
LDC   integer,1
LDC   integer,2
IX2   T8    ;array descriptor for ''a[@1..2]''
LDC   integer,1
CVT   integer,T8      ;blank descriptor for constant array
LDC   integer,1
LDC   integer,2
IX2   T8    ;select subrange
LDI   T8    ;load ''a[@1..2]''
ADD   T8
STO   T8
```

Because the operands to all Parallel P-code instructions must be the same type, when two array segments (that is, arrays or subsets of arrays) with the same shape but different types are combined, one must be converted to conform to the other one. For example, given the Parallel Pascal statements:

```
var
    a: array [1..5] of integer;
    b: array [5..9] of integer;

    a := a + b[@5..9];
```

the Parallel P-code definitions might be:

```
.ARRAY    T5,integer,1,1,5
.ARRAY    T6,integer,1,5,9
.LOCAL    1,T5,0
.LOCAL    2,T6,0
```

The two arrays would be loaded onto the stack by constructing their array descriptors and performing a LDI:

```
LLA   0,1        ;array descriptor for ''a''
IXO   T5
LLA   0,1        ;array descriptor for ''a''
IXO   T5
LDI   T4         ;load ''a''
LLA   0,2        ;array descriptor for ''b''
IXO   T6
LDI   T6         ;load ''b''
```

However, before these two arrays can be added (and the result stored), it is necessary to convert them to the same type. For example, the top-of-stack can be converted from type T5 to T4:

```
CVT   T6,T5
```

The array is converted in temporary memory and the array descriptor for the result replaces the array descriptor on top of the stack. (Note that the resulting descriptor is not ''blank'' - that is, the index ranges are filled in by CVT. ''Blank'' indexing information only results when a scalar is expanded to an array.)

In some cases, such as the example above, the index range of the array to be converted does not fall within the index range of the type it is converted to. In these cases, the implementation of CVT must adjust the index ranges so that they fall within the dimensions of the result type. (If a dimension of the result type is not large enough to contain a dimension of the operand an error has occurred, for in this case the two array operands can not possibly have the same shape.) After the operands have been converted to identical types, the arrays may be added and the result stored:

```
ADD  T5
STO  T5
```

At times, two stack operands will have the same type but the index ranges of the two operands will be different. The implementation must ''shift'' one of the operands so that the active array elements ''line up''. The choice of which operand to shift is left to the implementation except in the case of a STO instruction; in this case the data being stored must be aligned with the subset of the array it is to be stored into.

Finally, if scalar indexing is used the logical shape of the array is altered. Thus, given the definition:

```
var
    a: array [1..5,1..5] of integer;
```

which might be defined in Parallel P-code by

```
.ARRAY    T8,integer,2,1,5,1,5
.LOCAL    1,T8,0
```

then the statement

```
a[1,] := a[,1];
```

would be translated into Parallel P-code as

```
LLA  0,1
LDC  integer,1
.ARRAY    T9,integer,1,1,5
IX1  T8,T9      ;index by scalar -- note type conversion
IX0  T9    ;select entire dimension
LLA  0,1
IX0  T8    ;select entire dimension
LDC  integer,1
IX1  T8,T9      ;select entire dimension -- note type conversion
LDI  T9    ;load column
STO  T9    ;store in row
```

This illustrates the difference between the static (logical) type and the dynamic (physical) information which is contained on the run-time stack. Clearly the data allocations for the two ''T9'' types are non-identical; however, their logical types are identical and hence the two arrays are conformable.

The SEL instruction, like the IXI instruction, provides additional information to the dynamic descriptor (physical specification) and alters the static descriptor (logical type). For example, given the definition:

```
var
    a: array [1..5,1..5] of integer;
    r: array [1..5] of
         record
             x: array [1..5] of integer;
             y: real;
         end;
```

which might be represented in Parallel P-code as

```
.ARRAY    T8,integer,2,1,5,1,5
.ARRAY    T9,integer,1,1,5
.RECORD   T10,x,nil,T9
.RECORD   T10,y,nil,real
.ARRAY    T11,T10,1,1,5

.LOCAL    1,T8,0
.LOCAL    2,T11,0
```

then the statement

```
a := r.x;
```

would be translated into Parallel P-code as

```
LLA  0,1  ;''blank'' descriptor for ''a''
IX0  T8
IX0  T8   ;descriptor for entire array ''a''

LLA  0,2
IX0  T11  ;descriptor for array of entire records
SEL  T11,x,T8
IX0  T8   ;descriptor for entire array ''x'' in ''r''
```

The physical storage corresponding to all elements of the array ''a'' is clearly different than the storage corresponding to all elements of the field ''x'' the array of records ''m''. However, their logical types are identical and hence the two items are conformable.

4.5 Standard Functions and Procedures

Parallel Pascal, like standard Pascal, provides a set of standard functions and standard procedures to perform tasks which are difficult or impossible to specify directly. Standard functions and procedures can in some sense be considered as extended operators, for the types of (and often even the number of) their parameters may vary.

In P-code, the arguments to a standard function or procedure are loaded onto the stack and the routine is called. At the P-code level all standard procedure calls have a fixed number of arguments and a fixed type. When a Pascal procedure such as ''write'' has multiple arguments of different types it is implemented as a series of calls to fixed-format routines such as

''wri'' (write integer), ''wrr'' (write real), etc.

In Parallel Pascal some standard functions may be called with a variable number of arguments which cannot be serialized into a set of fixed-format calls. An example is the ''shift'' function, for which the number of dimensions of the first argument (the array to be shifted) determines the number of parameters which are passed to the function.

To deal with the variable number of arguments and the varying types of the arguments (since arrays of any shape may be operated upon) Parallel P-code uses a modified calling sequence. First, the stack is marked with the MST instruction. Standard functions and procedures are considered to be at lexical level zero; no other routine are (the outermost block - the program block - is at lexical level 1). The arguments are then computed. Scalars are treated as in standard P-code: if passed ''by value'' a scalar expression is evaluated; otherwise, the address of the scalar is passed. Arrays and records are always passed as descriptors. If they are passed ''by value'' an LDI is performed, so that the descriptor points to a temporary-storage copy of the data, rather than to the original variable. If they are passed ''by reference'' the original descriptor is passed. Finally, the CSP instruction is used to call the standard procedure or function. If the called routine is a function it is responsible for storing its result on top of the runtime stack when it exits; in all cases the return from the called routine

will reset the stack back to the marked location.

Standard procedures and functions operate principally upon an item of a particular data type; the other arguments have fixed data types. For instance, the ''shift'' routine operates upon an array whose type and shape may vary; the additional operands are all integers. Parallel P-code provides a mechanism for specifying the logical data type of this primary argument as well as the result type of the function. Thus, the format of the CSP instruction is:

```
CSP  stdfunc,argtype,restype
```

where ''stdfunc'' is the name of the standard function or procedure, ''argtype'' is the (logical) data type of the primary argument, and ''restype'' is the (logical) data type of the function result. (If the called routine is a standard procedure the literal string ''nil'' will be used.)

4.6 User-defined Functions and Procedures

Unlike the standard functions, a user-defined function or procedure is always called with a fixed number of arguments whose types are constant. This leads to a regular structure for calling user-defined routines

As discussed earlier, standard P-code and Parallel P-code both organize memory allocation on the run time stack into stack frames. Each stack frame contains the static and dynamic links,

the return address, the arguments to the routine, and the local variables. (In the case of Parallel P-code these variables are symbolically specified.)

In standard P-code, a subroutine or function call involves several steps. First, the stack is ''marked''. This sets up a new stack frame and fills in the static and dynamic links. Next, each argument to the routine is generated. For arguments passed ''by value'' this involves the evaluation of the argument as an expression on the stack; for arguments passed ''by reference'' this consists of loading the address of the argument onto the stack. (Standard P-code has no provision for passing procedure or function parameters; these are therefore not implemented in the P4 compiler.) After all of the arguments have been prepared the function is called. When the function returns, the stack is reset back to the marked location. If the called routine was a function, the function result is left on top of the runtime stack.

The calling sequence in Parallel P-code is very similar to the standard P-code case. The stack is marked with the MST instruction, which specifies the lexical level of the procedure or function to be called. The arguments are then computed. Scalars are treated as in standard P-code: if passed ''by value'' a scalar expression is evaluated; otherwise, the address of the scalar is passed. Arrays and records are always passed as array descriptors. If they are passed ''by value'' a LDI is performed, so that the descriptor points to a temporary-storage copy of the

array or record, rather than to the original variable. If they are passed ''by reference'' the original descriptor is passed. Parallel P-code, like standard P-code, does not provide for passing functions or procedures as parameters.

The function or procedure is called with the CUP instruction, which has the syntax:

```
CUP  level,routinename,resulttype
```

where ''level'' is the lexical level of the called routine, ''routinename'' is its name, and ''resultype'' is the data type of the function result. (If the called routine is a procedure this will be the literal string ''nil''.)

The function return is performed by the RET instruction. This instruction has the syntax:

```
RET  type
```

where ''type'' is the type of the function result. In the case of procedures, ''type'' will be the literal string ''nil''. Local variable 0 is used by functions to hold the function result. When the RET instruction is executed, the stack is ''popped'' back to the caller, and the function result (if the called routine was a function) is left on top of the runtime stack. If the result was an array or record the top-of-stack will be an array descriptor.

4.7 Conditional Execution

Parallel Pascal, like standard Pascal, provides a set of control flow constructs. In standard P-code, these are implemented with four ''jump'' instructions: XJP, FJP, UJP, and UJC. The implementation of the if, case, for, while, and repeat-until statements in Parallel P-code is identical to the implementation in standard P-code. The XJP and FJP instructions use the top of the runtime stack as an operand - as an index into a table of addresses (XJP), or as a Boolean condition (FJP or ''jump if false''), and in both of these cases the quantity on the stack must be a scalar.

Parallel Pascal also provides the where statement to specify conditional assignment of expressions to arrays, controlled by an array expression. The where statement cannot be (efficiently) implemented with the scalar-oriented control mechanisms of standard P-code.

Parallel Pascal specifies that array assignments are conditional within the body of a where statement. Thus, the implementation in Parallel P-code will only affect the STO operator (the STO operator is never used to store array data).

The MPP and most SIMD-class parallel processors associate with each processor a flag known as the ''mask bit'' or ''activity bit''. This bit controls whether or not the processor is enabled or disabled. The collection of mask bits for each processor can be considered to be a ''mask array''.

This array can further be considered to be a Boolean array, since the values it contains are binary. The controlling expression of a where statement in Parallel Pascal is a Boolean array; hence, it is natural to implement the where statement by using this array as a mask array.

Parallel Pascal permits where statements to be nested. All of the mask arrays in such a nested collection of statements must have the same shape. Thus, there is a need for a nested sequence of conditional expressions.

The current conditional status of a set of N nested conditionals can be determined by using a stack of length N (bits). If the current conditional state is A, and a where statement is encountered which evaluates to B, the new conditional state is AB (the Boolean product of A and B). At some later point, if an otherwise is encountered, the desired conditional state is $A\overline{B}$. Taking ''+'' as the symbol for a Boolean ''inclusive or'' and ''$\diamond$'' as the symbol for a Boolean ''exclusive-or'', and recalling:

$$x \diamond y \equiv x\overline{y} + \overline{x}y$$

$$\overline{xy} \equiv \overline{x}+\overline{y}$$

then

$$AB \diamond A \equiv (AB)\bar{A} + (\overline{A\bar{B}})A \equiv A\bar{A}B + (\bar{A}+\bar{B})A$$

$$\equiv (A\bar{A})B + A\bar{A}+A\bar{B} \equiv A\bar{B}$$

Using this result, the stack-oriented implementation can be defined.

Initially the stack is empty, and all processors are enabled. When a where conditional is encountered, a Boolean ''and'' is performed with the current top of the stack (if the stack is non-empty) and the result is pushed onto the stack. When an otherwise is encountered, a Boolean ''exclusive or'' is computed between the top two elements of the stack and the result replaces the top of the stack. (If the stack contains only one item, a Boolean ''not'' - complement - is performed.) When the end of the conditional is encountered, the stack is popped.

The implementation of nested conditional masks in Parallel P-code is based upon this algorithm. A new conditional expression is pushed onto the mask stack by the WHR instruction, which has the syntax:

WHR type

The sense of the most recent conditional is reversed with the OTW instruction, which has the syntax:

OTW type

and the mask stack is ''popped'' by the ENW instruction, which

has the syntax:

ENW type

When a new level of masking is entered, the mask expression is computed on the runtime stack, and then the WHR statement is used to put the mask into effect. If the implementation so desires, the WHR instruction may optionally not pop this expression off of the run time stack, provided that the ENW instruction does. This retains the temporary memory which holds the mask expression. If a set of nested conditionals are evaluated, there will be a set of temporary mask array descriptors on the run time stack. The implementation may then use the storage to which these refer to implement the mask stack. (This method requires a small amount of temporary memory allocated outside of the run time stack which holds pointers to all of the descriptors on the stack.)

Parallel Pascal specifies that the effect of a mask is not transmitted to a called function or procedure. Thus, each procedure or function has its own mask stack. The implementation may choose to include the information about the mask stack for each routine in the stack frame (e.g. along with the static and dynamic links, etc.).

4.8 The ''with'' Statement

Both standard Pascal and Parallel Pascal provide the with statement to reduce the need to fully specify record accesses. For instance, the following sequence of code:

```
recp↑.subrec.x := 0;
recp↑.subrec.y := 0;
recp↑.subrec.z := 0;
```

could be written as

```
with recp↑.subrec do
  begin
    x := 0;
    y := 0;
    z := 0;
  end;
```

The use of a with statement has two principal advantages. The first advantage is that the program notation is simplified by eliminating repetitions of the same record specification. (This is not always an advantage, however, in programs with many record types, because at times it becomes difficult for a human to keep track the record to which each component refers.) The second advantage is that the address of the record to which the fields belong is calculated only once, rather than each time a field is referenced. For instance, in the first example above (without the with statement) the pointer ''recp'' is de-referenced three times. When the with statement is used (the second example above), the address computation is performed only once.

To determine the implementation in Parallel P-code, it is useful to first consider the implementation in standard P-code.

Two types of records are used as operands to a with statement - those which are ''normal'' variables and those which are specified by a record pointer.

In the case of non-pointer variables, the (lexical) address of the specified record is known at compile time. Whenever a reference is made to a field in that record the compiler adjusts this address by the offset of the specified field and loads that address on the stack. Thus, the (lexical) address of every field specified in the with statement is known at compile time if the argument to the with is an ordinary variable.

When a pointer is used in a with statement, the value of the pointer must be preserved so that references to fields in the record indicated by that pointer can be properly addressed. In this case, not even the address of the pointer itself is necessarily known at compile time. For instance, given the following code segment:

```
type
    rec = record
              x: integer;
              y: real;
          end;
    ptr = ↑rec;
    pptr = ↑ptr;

var
    pp: pptr;

begin
    new(pp);    (* "pp" points to an object of type "ptr" *)
    new(pp↑);   (* "pp↑" points to an object of type "rec" *)
    with pp↑↑ do
        x := 0;
    ...
```

C-2

the address of the pointer itself (``pp↑'' is clearly not known at compile-time). Hence, it is necessary to compute the value of the pointer and save it in a temporary stack location. The P4 compiler ``knows'' the size of each stack element and can compute the address of the temporaries. When a field within an applicable record is referenced, the pointer is loaded onto the top of the stack and the offset (in this case, the offset of ``x'' relative to the start of the record) is added. This produces the effective address of the desired operand; it may then be used as the address for a LDI or STO.

In Parallel P-code, the situation is somewhat different because the ``front end'' of the compiler is unaware of the layout of records. A compile-time-constant (lexical) address for an ``ordinary'' record (that is, one which is not accessed through a pointer) cannot be calculated. (Actually, the compile-time address is known if the record in question is not itself a component of another record.) In addition, the expression may be an array of records rather than a single record. For this reason, the Parallel Pascal compiler always calculates the value of the address expression in a with statement. The descriptor for the record is loaded (either by construction from the compile-time-known lexical address and a sequence of SEL instructions, or - in the case of a pointer - by loading the ``pointer''). This descriptor is then stored in a temporary variable. When a record field is used, the descriptor is loaded from the temporary variable, the necessary SEL is

performed to access the desired field, and the resulting descriptor is used.

Unlike standard P-code, which stores only the address of the record, Parallel P-code must store the entire descriptor. (Recall that descriptors in Parallel P-code play the same role as pointers in standard P-code.) This presents a problem, because several different descriptors may be required at different times during the execution of a routine. It is desirable to overlay the space which is allocated to them as much as possible. The need for this sharing of temporary storage contributed to the syntax of the .LOCAL pseudo operator, described above.

Briefly, the syntax of the .LOCAL pseudo operator is

```
.LOCAL    index,type,overlay
```

where ``index'' is an index that symbolically identifies the location in the current procedure activation, ``type'' is the data type, and ``overlay'' is used for memory sharing. The ``index'' and ``type'' fields are described in more detail above; the ``overlay'' field is of interest here.

Local variables are allocated according to the following rule: if ``overlay'' is zero, allocate the new variable beginning at the next available location; otherwise, allocate the new variable beginning at the same location as the variable whose index is ``overlay''. Thus, the following statements:

```
.LOCAL    1,integer,0
.LOCAL    2,real,0
.LOCAL    3,integer,2
```

causes local variables 2 and 3 to be allocated in the next available memory after local variable 1. If possible, variables 2 and 3 are to share the same memory. (The implementation is free to ignore this memory sharing specification. This may be necessary if the variables would reside in different memories - something which is not the case for descriptors. However, variables which are defined to use disjoint memory must indeed be allocated that way; otherwise unexpected memory sharing will result.)

4.9 References

1 K. V. Nori, U. Ammann, K. Jensen, and H. Naegeli, The Pascal (P) Compiler - Implementation Notes, Institut fur Informatik, Eidgenoessische Technische Hochschule, Zurich (1975).

2 Philip A. Nelson, "A Comparison of PASCAL Intermediate Languages," ACM SIGPLAN Notices Vol. 14(8), pp.208-213 (August 1979).

3. " University of Illinois internal report..

4 Mary E. Zosel, "A Modest Proposal for Vector Extensions to ALGOL," ACM SIGPLAN Notices Vol. 10(3), pp.62-71 (March 1975).

5 R. G. Zwakenberg, "Vector Extensions to LRLTRAN," ACM SIGPLAN Notices Vol. 10(3), pp.77-86 (March 1975).

6 Alfred V. Aho and Jeffrey D. Ullman, Principles of Compiler Design, Addison-Wesley, Reading, Massachusetts (1977).

5: MEMORY MANAGEMENT

5.1 The Memory Problem

The language Parallel Pascal was defined in section 2, and an implementation using Parallel P-code was described in section 3. The language definition for Parallel Pascal places no limits upon the utilization of memory by a program. Similarly, Parallel P-code has no inherent restrictions on the size or shape of arrays, whether they are parallel or not.

Although the specifications of the high-level and intermediate-level languages contain no size restrictions, the first implementations of Parallel Pascal almost certainly will. The implementation of the language is affected by two fundamental hardware constrants: the size of the processor array and the amount of memory in each processing element's local memory. In this section, these problems will be considered relative to the implementation of Parallel Pascal on the Massively Parallel Processor[1].

The first restriction arises from the fact that the MPP has a 128×128 element processor array. If data arrays are declared

with these dimensions there is no problem; however, arrays with much larger or much smaller dimensions require special consideration. The problem of assigning memory locations to variables is examined in section 5.2.

The second implementation restriction results from the very small local memory in each processing element. Each PE has only 1024 bits of random-access memory. Although this amounts to two megabytes of memory for the whole array, it is limited when manipulating large amounts of data - e.g. each local memory can store only 32 (32-bit) floating-point numbers. The main memory is supplemented by two levels of secondary memory: a high-speed random-access memory called the ''staging buffer'', and a secondary input-output system (connection to the host computer or a proposed parallel disc system). The staging buffer in the initial delivered version of the MPP will have a two megabyte capacity; plans are underway to eventually expand to sixteen megabytes, out of a total capacity of 64 megabytes. Section 5.3 examines the efficient management of memory with this configuration.

5.2 Memory Layout

5.2.1 Introduction

In this section, the allocation of storage on the PE array is considered. To facilitate discussion, the following definitions are assumed:

NROW = number of rows in the processor array
NCOL = number of columns in the processor array

An array whose last two dimensions have sizes NROW and NCOL, respectively, can be easily mapped into the parallel array memories - if the array is declared as:

```
var
    arr: parallel array [I..J,K..L] of something;
```

where NROW=J-I+1 and NCOL=L-K+1, then ``arr[i,k]'' will be stored in row i-I, column k-K.

This storage mapping can be extended to multi-dimensional arrays whose last two index ranges have sizes NROW and NCOL, respectively. The address within an array memory is computed according to the usual formula. As an example, given the definition:

```
var
    arr: parallel array [9..12, 4..5, 1..128, 8..135] of integer;
```

and assuming that the base address within the PE memories for ``arr'' is $a_0$ and that integers are stored in 16 bitplanes, then ``arr[i,j,m,n]'' will be stored in row m-1, column n-8, at address

$$a_0 + 16\times[(i-9)\times(5-4+1) + (j-4)]$$

When the last two dimensions of an array do not match the size of the PE array another mapping strategy must be used. There are two cases to be considered - data array dimensions smaller than the PE array size, and data array dimensions larger than the PE array size.

5.2.2 Small Arrays

If an array dimension is smaller than the corresponding dimension of the PE array, then some PE's will not be used to store the array. For instance, a 64×64 array will only use one quarter of the PE's in the MPP. This manifests itself in two ways. First, it is extremely wasteful of the main memory (a very precious commodity). Second, since the edges of the array no longer coincide with the edges of the PE array, rotating data through the array will require more than one cycle per position rotated.

One possible way to store a small array would be to use a contiguous subsection of the hardware array; e.g. to store a 32×32 array on the MPP one could use all PE's (i,j), $0\le i<32$, $0\le j<32$. This implementation works well if only elemental operations are performed upon the data. In addition, the ``shift'' function can be used without substantial overhead (an extra cycle is required for each shift from the ``east'' or ``south'' in order to force the incoming values to zero). However, the ``rotate'' function will be very expensive because

it will be necessary to propagate data shifted out one end across 96 inactive PE's to reach the other end of the array.

An alternate storage scheme would be to store in every fourth PE in each direction; that is, the 32×32 array described above could store array element (i , j) in PE (i×4 , j×4) (because 128/32 = 4). This scheme has the advantage that data can be rotated across the logical array without having to propagate edge information across a large number of inactive PE's. However, each position shifted now requires 4 hardware shift operations instead of one because the data items are further apart. In addition, this scheme works best when the data array has a dimension which evenly divides the length of the hardware array; the implementation is not as apparent if the data array is, say, 59×59. Finally, if arrays are stored in non-contiguous PE's then it will be necessary to compress or expand them when subsets interact with subsets of arrays of different sizes. For these reasons, it seems advisable that small arrays be stored in contiguous subsets of the hardware array.

Since small arrays do not occupy every memory module in the PE array, several small arrays can share the same memory offset. For instance, four 64×64 arrays can be stored at the same PE memory address in the 128×128 MPP array. Arrays of different sizes can also be stored together - the available PE's in a memory plane can be ``parceled out'' as required.

If a small array has more than two dimensions, one possible

way of storing the array is to store several subarrays in the same memory plane(s). For instance, a 4×64×64 array could be stored at the same address in the 128×128 MPP memories. This storage scheme has the advantage that all elements of the array can be operated upon at once. The disadvantage of this scheme is the time required to transfer data from element (i,j,k) to element (i+1,j,k) - in the case described above 64 shifts are required; whereas if the storage were ''vertical'' the data could be handled internally by the PE's.

A special case of the general problem of small arrays is the handling of vectors. Although a vector has only one dimension, it is frequently convenient to manipulate a vector in the PE array. In addition, a vector may result from a reduction operation upon an array (e.g. using ''sum'' along the columns of a matrix). A reasonable storage implementation would be to treat a vector as a matrix with only one row or one column (whichever is more convenient for the problem at hand). Like other small arrays, several vectors could then be stored in the same memory plane (e.g. in successive rows of the PE array).

Finally, if a matrix or vector is very small the implementation may wish to ignore the parallel specification and store the array in the scalar memory. Small vectors and matrices can easily be accomondated there without incurring a significant storage overhead. If the array is small, not much parallelism will be sacrificed by performing operations with it serially

instead of in parallel.

The ability to specify the data storage format for small arrays is lacking in Parallel Pascal. The decision about storage formats is therefore left to the code generator. A future language revision might include some provision for specifying the data storage, along the lines of the parallel keyword which Parallel Pascal does provide.

5.2.3 Large Arrays

Large arrays present a different set of problems which must be addressed. First, since the PE array is smaller than the data array, no mapping of the last two array dimensions into the PE array will be one-to-one. Instead, some PE's will contain more than one point. Second, large arrays seriously impact the main memory capacity - if the array is too large it may not fit in the main memory at all. An example of this case is a 2048×2048 array of integers, which requires eight megabytes of main memory (the MPP as delivered will have only two megabytes of main memory). Finally, if the data array size is not an even multiple of the PE array size, extra operations will be required when data rotations are performed. This last case closely resembles the rotation problem for small arrays which was discussed in the previous section; it will not be considered further here.

The first problem is the manner in which large arrays are to be stored. For convenience, the last two dimensions of the data array will be assumed to be multiplies of the PE array

dimensions. In the following discussion, two-dimensional arrays are considered for convenience; additional dimensions are implemented ''vertically'' within the PE array and are therefore of no special interest here.

Let the PE array have dimensions (M , N) (both M and N are 128 on the MPP) and the data array have dimensions (A×M , B×N). There are many possible ways in which to map the large data array into the PE array, but simplicity dictates that the array be stored in such a way that each PE is associated with A×B elements.

If the data array is fairly small relative to the main memory of the matrix processor (e.g. a 256×256 array of 8-bit data on the MPP) one possible implementation is to store in each PE memory a M×N subimage. That is, if ''arr'' were defined by:

```
(* NROW = A*M, NCOL = B*N *)
var
     arr: parallel array [0..NROW, 0..NCOL] of integer;
```

and it were stored starting at a_0, then (assuming an integer is 16 bits) arr[i,j] would be stored in PE

$$\left(\left\lfloor\frac{i}{A}\right\rfloor , \left\lfloor\frac{j}{B}\right\rfloor\right)$$

at address

$$a_0 + 16\times[(i \bmod A)\times B + (i \bmod B)]$$

The advantage of this storage scheme is that adjacent points are often in the same PE and no data transfers are needed. This can be very useful when operations are performed involving near

neighbors. The disadvantage of this scheme is that when a section of the array is operated upon (e.g. a 128×128 piece of a 256×256 array) the parallelism will be lower. Also, large arrays cannot be conveniently manipulated because they will not fit into main memory.

An alternate storage method is to divide the large array into ''chunks'', each of which is the same size as the PE array. Given the array ''arr'' defined above, this scheme would map arr[i,j] (with base address a_0) into PE

$$(i \bmod M\ ,\ j \bmod N)$$

at address

$$a_0 + 16\times\left[\left\lfloor\frac{i}{M}\right\rfloor\times B + \left\lfloor\frac{j}{N}\right\rfloor\right]$$

The advantage of this scheme is its capability for performing operations on subsections of the array with the maximum degree of parallelism. This facilitates the processing of large arrays (which are too large for the PE memories).

Another important factor in the choice of a storage layout for large data arrays is the ease with which the arrays can be transferred into and out of the main memory. In general, the second format (breaking a large array into pieces) is easier to handle than the first format (storing a local subimage in each memory); however, on the MPP both formats can be handled - the staging memory is capable of ''crinkling'' an input image in

order to store a local window in each PE.

The initial implementation of Parallel Pascal on the MPP will restrict the size of the last two dimensions of a parallel array to be less than or equal to the PE array size. As a result, the programmer will have to explicitly deal with large data arrays in one of the ways described above (i.e. storing multiple points in each PE or dividing the array into ''chunks''). The latter is performed by dimensioning an 128N×128M array as

```
type
    large = parallel array [1..N,1..M,1..128,1..128] of integer;
```

and logically associating the first and third dimensions of this array with the first dimension of the large array (similarly, the second and fourth dimensions of the declared array are associated with the second dimension of the large array). Simple operations on the array can be expressed directly; statements such as

```
a := b + c;
```

mean the same thing regardless of how the array is dimensioned. However, data movements in the large array must be mapped into movements in the small array. The following two routines, ''lshift'' and ''lrotate'' illustrate how a shift and rotate on a (logical) large 128N×128M array would be implemented on an N×M×128×128 array. These functions also illustrate how an automatic memory allocation scheme would handle large arrays which are stored in ''chunks''.

```
(*
 *   LSHIFT - Shift large array
 *
 *  The array "a" (conceptually size N*MPPROW by M*MPPCOL,
 *  dimensioned as a[1..N,1..M,1..MPPROW,1..MPPCOL]) is end-off
 *  shifted.
 *
 *  The shifting is done in two stages - by rows and then by columns.
 *)

const
    MPPROW = 128;    (* number of MPP rows *)
    MPPCOL = 128;    (* number of MPP columns *)
    N = 10;          (* N*MPPROW = number of conceptual rows *)
    M = 20;          (* M*MPPCOL = number of conceptual columns *)

type
    MPPREAL = parallel array [1..MPPROW,1..MPPCOL] of real;
    MPPBOOL = parallel array [1..MPPROW,1..MPPCOL] of Boolean;
    LARRAY = array [1..N, 1..M] of MPPREAL;

function lshift(a: LARRAY; r,c: integer) : LARRAY;
var
    bsr: 0..N;              (* block shift amount (rows) *)
    isr: 0..MPPROW;         (* internal shift amount (rows) *)
    bsc: 0..M;              (* block shift amount (cols) *)
    isc: 0..MPPCOL;         (* internal shift amount (cols) *)
    tmp: LARRAY;            (* temporary array *)
    mask: MPPBOOL;          (* mask for internal rotates *)

begin

    bsr := r div MPPROW;
    isr := r mod MPPROW;
    bsc := c div MPPCOL;
    isc := c mod MPPCOL;

    tmp := rotate(a, 0, 0, isr, isc);

    mask := shift(a=a, 0, 0, isr, 0);
    where not mask do
        tmp := shift(a, 1, 0, 0, 0);

    mask := shift(a=a, 0, 0, 0, isc);
    where not mask do
        tmp := shift(a, 0, 1, 0, 0);

    lshift := shift(tmp, bsr, bsc, 0, 0);

end;
```

```
(*
 *    LROTATE - Rotate large array
 *
 *   The array "a" (conceptually size N*MPPROW by M*MPPCOL,
 *   dimensioned as a[1..N,1..M,1..MPPROW,1..MPPCOL]) is rotated
 *   (circularly shifted).
 *
 *   The rotation is done in two stages - by rows and then by columns.
 *)

const
    MPPROW = 128;     (* number of MPP rows *)
    MPPCOL = 128;     (* number of MPP columns *)
    N = 10;           (* N*MPPROW = number of conceptual rows *)
    M = 20;           (* M*MPPCOL = number of conceptual columns *)

type
    MPPREAL = parallel array [1..MPPROW,1..MPPCOL] of real;
    MPPBOOL = parallel array [1..MPPROW,1..MPPCOL] of Boolean;
    LARRAY = array [1..N, 1..M] of MPPREAL;

function lrotate(a: LARRAY; r,c: integer) : LARRAY;
var
    bsr: 0..N;              (* block rotation amount (rows) *)
    isr: 0..MPPROW;         (* internal rotation amount (rows) *)
    bsc: 0..M;              (* block rotation amount (cols) *)
    isc: 0..MPPCOL;         (* internal rotation amount (cols) *)
    tmp: LARRAY;            (* temporary array *)
    mask: MPPBOOL;          (* mask for internal rotates *)

begin

    bsr := r div MPPROW;
    isr := r mod MPPROW;
    bsc := c div MPPCOL;
    isc := c mod MPPCOL;

    tmp := rotate(a, 0, 0, isr, isc);

    mask := shift(a=a, 0, 0, isr, 0);
    where not mask do
        tmp := rotate(a, 1, 0, 0, 0);

    mask := shift(a=a, 0, 0, 0, isc);
    where not mask do
        tmp := rotate(a, 0, 1, 0, 0);

    lrotate := rotate(tmp, bsr, bsc, 0, 0);

end;
```

5.3 Data Migration

Because the main memory on the MPP is so small, it is inevitable that many programs will require more memory than can be provided in the processor array alone. Thus, some form of data migration will be required. This can be implemented in one of two ways. First, the programmer could be required to handle all data migration. Second, an automatic memory management system could be used and the programmer could be unaware of the transfer of data between memories.

In the following sections, the behavior of the MPP is considered in an attempt to determine an appropriate memory management strategy.

5.3.1 The Overlap Factor

The movement of data between memories will slow down the computation of the MPP system on a problem by some amount. On the MPP, input-output and computations may be overlapped. A quantity which is of some interest is the execution time penalty for not overlapping input-output with computation. Let T_c be the total computation time, T_{io} be the total input-output time, and T_o be the time required if input-output is overlapped with computation as much as possible. Then f, the fraction of the possible speed obtained with non-overlapped input and output versus a fully-overlapped implementation, may be defined as

$$f = \frac{T_o}{T_{io}+T_c}$$

T_o is bounded by the inequalities:

$$T_o > T_{io} \quad , \quad T_o > T_c \quad , \quad T_o < T_{io} + T_c$$

The last of these implies that $f<1$. The first two inequalities bound f from below:

1) if $T_c > T_{io}$ then $T_o = T_c \Rightarrow f = \frac{T_c}{T_{io}+T_c} > \frac{T_c}{T_c+T_c} = 1/2$

2) if $T_{io} > T_c$ then $T_o = T_{io} \Rightarrow f = \frac{T_{io}}{T_{io}+T_c} > \frac{T_{io}}{T_{io}+T_{io}} = 1/2$

3) if $T_c = T_{io}$ then $T_o = T_c = T_{io} \Rightarrow f = \frac{T_o}{T_o+T_o} = 1/2$

In summary, $1/2 \leq f \leq 1$. Thus, the maximum penalty for not overlapping input-output and computation is a speed reduction of 1/2.

5.3.2 I/O-CPU Time Ratios

Another factor of interest is the relationship between the time required for the input-output associated with a problem and the CPU time required to solve that problem.

Let S be the execution speed (in operations per second). Define T_{op} as the time required to perform one array operation. Since the MPP has an array of 128×128 PE's, T_{op} can be calculated by

$$T_{op} = \frac{128 \times 128}{S} = \frac{16384}{S}$$

The MPP has a two-level secondary memory. Data is transferred from the PE memories to the staging buffer by shifting it across the rows of the array. It requires 128 cycles

to shift a single bitplane into the array. Let T_{cpu} be the CPU cycle time and B be the number of bitplanes to be shifted. Define T_α as the time required to transfer a data item from the staging buffer to the PE memories. Then

$$T_\alpha = 128BT_{cpu}$$

One possible configuration of the MPP is to use a very-high-speed parallel disc system for secondary storage. Let L_d be the average seek and rotational latency of the disc, and R be the transfer rate (in bytes per second). Define T_β as the time required to transfer data from the disc to the staging area. Then

$$T_\beta = L_d + \frac{128\times128\times B}{8}\times\frac{1}{R} = L_d + 2048\frac{B}{R}$$

(The number of bytes is equal to the number of bitplanes divided by 8.)

Define X as the ratio of the time spent on input and output to the time spent performing the operation. Then

$$X = \frac{T_{io}}{T_{op}} = \frac{T_\alpha+T_\beta}{T_{op}} = \frac{128BT_{cpu}+L_d+2048\frac{B}{R}}{\frac{16384}{S}} = \frac{SBT_{cpu}}{128} + \frac{SL_d}{16384} + \frac{SB}{8R}$$

The first term represents the contribution of the staging area; the second term is due to the disc access latency, and the third term is due to the disc transfer rate.

The value of X represents the average number of operations per array element required to keep the PE array busy. If, instead, only αX operations are performed (where $0<\alpha<1$) then the

average utilization of the PE array will be α. Thus, when X is large the input-output will dominate unless the task is very processor-intensive.

X can now be computed for three typical MPP operations: 8-bit integer addition (computing a 9-bit result), 32-bit floating-point addition, and 32-bit floating-point multiplication.

5.3.2.1 I/O-CPU time ratio: integer addition

The reported speed of the MPP performing 8-bit integer addition is 6553.6 million operations per second. There are two 8-bit operands and one 9-bit result. The cycle time of the MPP is 100 nanoseconds. Hence,

$$S=6553.6\times10^{6}$$
$$B=8+8+9=25$$
$$T_{cpu}=10^{-7}$$
$$X = \frac{(6553.6\times10^{6})(25)(10^{-7})}{128} + \frac{(6553.6\times10^{6})L_d}{16384} + \frac{(6553.6\times10^{6})(25)}{8R}$$
$$=128+(4\times10^{4})L_d+\frac{2.048\times10^{10}}{R}$$

If the data is transferred between the PE memories and the staging buffer, the input-output will take 128 times longer than the computation. If a secondary memory is involved, its latency must be very low and its transfer rate very high. Figure 1 shows the dependence of X upon the disc transfer rate for two values of

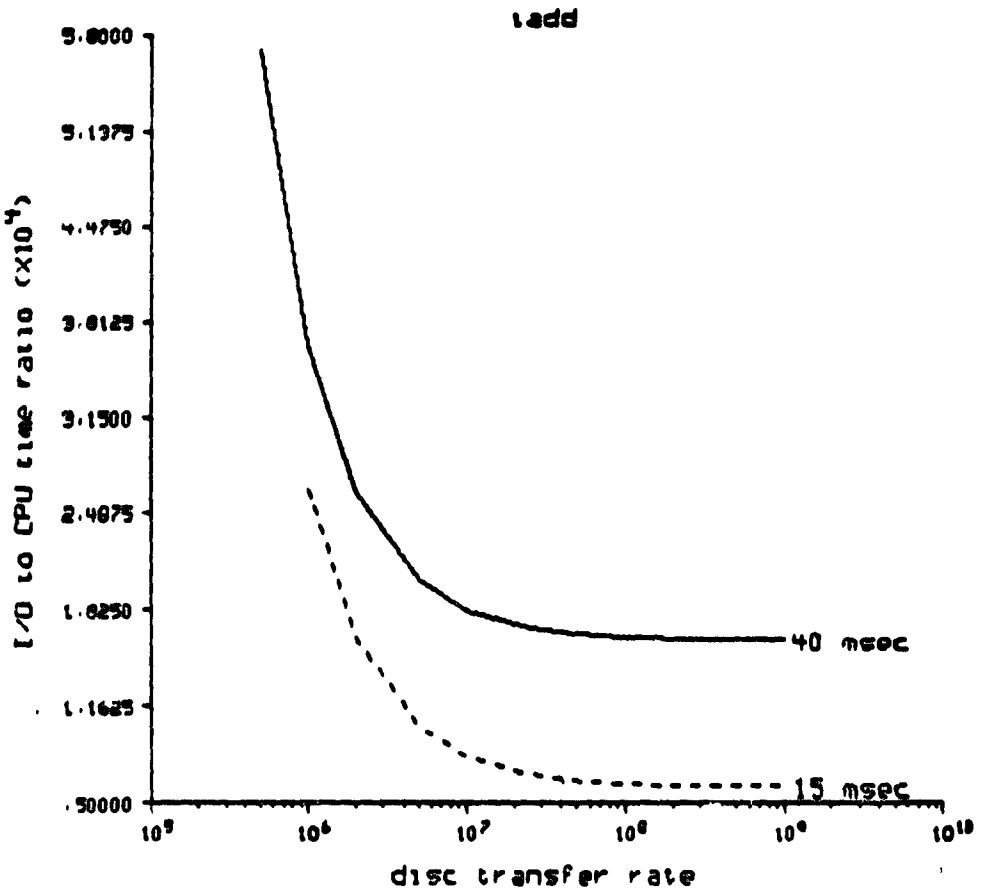


Figure 1: I/O-CPU Time Ratio: 8-bit addition

L_d.

For this relatively-simple operation, the processing time is swamped by the time required to perform input-output. For instance, a disc system with L_d=15ms (fast by current standards) and R=10^8Mb/s (very fast relative to current technology), X is still high:

$$X = 128+(4\times10^4)(.015)+\frac{2.048\times10^{10}}{10^8} = 6332.8$$

This means that in order to achieve 50% processor utilization when performing integer addition with two input arrays and one output array, it is necessary to perform (6332.8)(0.50) = 3166.4 array operations.

5.3.2.2 I/O-CPU time ratio: floating addition

The reported speed of the MPP performing 32-bit floating-point addition is 470 million operations per second. There are two 32-bit operands and one 32-bit result. The cycle time of the MPP is 100 nanoseconds. Hence,

$S=470\times10^6$
$B=32+32+32=96$
$T_{cpu}=10^{-7}$

Using these values, X can be computed as

$$X = \frac{(470\times10^6)(96)(10^{-7})}{128} + \frac{(470\times10^6)L_d}{16384} + \frac{(470\times10^6)(96)}{8R}$$

$$=35.25+(2.87\times10^4)L_d+\frac{5.64\times10^9}{R}$$

If the data is transferred between the PE memories and the

staging buffer, the input-output will take approximately 35 times longer than the computation. If a secondary memory is involved, its latency must be very low and its transfer rate very high. Figure 2 shows the dependence of X upon the disc transfer rate for two values of L_d.

While the input-output time is still much greater than the computation time, the difference is an order of magnitude less than in the 8-bit integer addition case above; for example, given L_d=15ms and R=10^8Mb/s,

$$X = 35.25 + (2.87\times10^4)(1.5\times10^{-2}) + \frac{5.64\times10^9}{10^8} = 522.15$$

5.3.2.3 I/O-CPU time ratio: floating multiplication

The reported speed of tne MPP performing 32-bit floating-point multiplication is 291 million operations per second. There are two 32-bit operands and one 32-bit result. The cycle time of the MPP is 100 nanoseconds. Hence,

$S=291\times10^6$

$B=32+32+32=96$

$T_{cpu}=10^{-7}$

Using these values, X can be computed as

$$X = \frac{(2.91\times10^8)(96)(10^{-7})}{128} + \frac{(2.91\times10^8)L_d}{16384} + \frac{(2.91\times10^8)(96)}{8R}$$

$$= 21.83+(1.776\times10^4)L_d+\frac{3.49\times10^9}{R}$$

If the data is transferred between the PE memories and the staging buffer, the input-output will take approximately 22 times

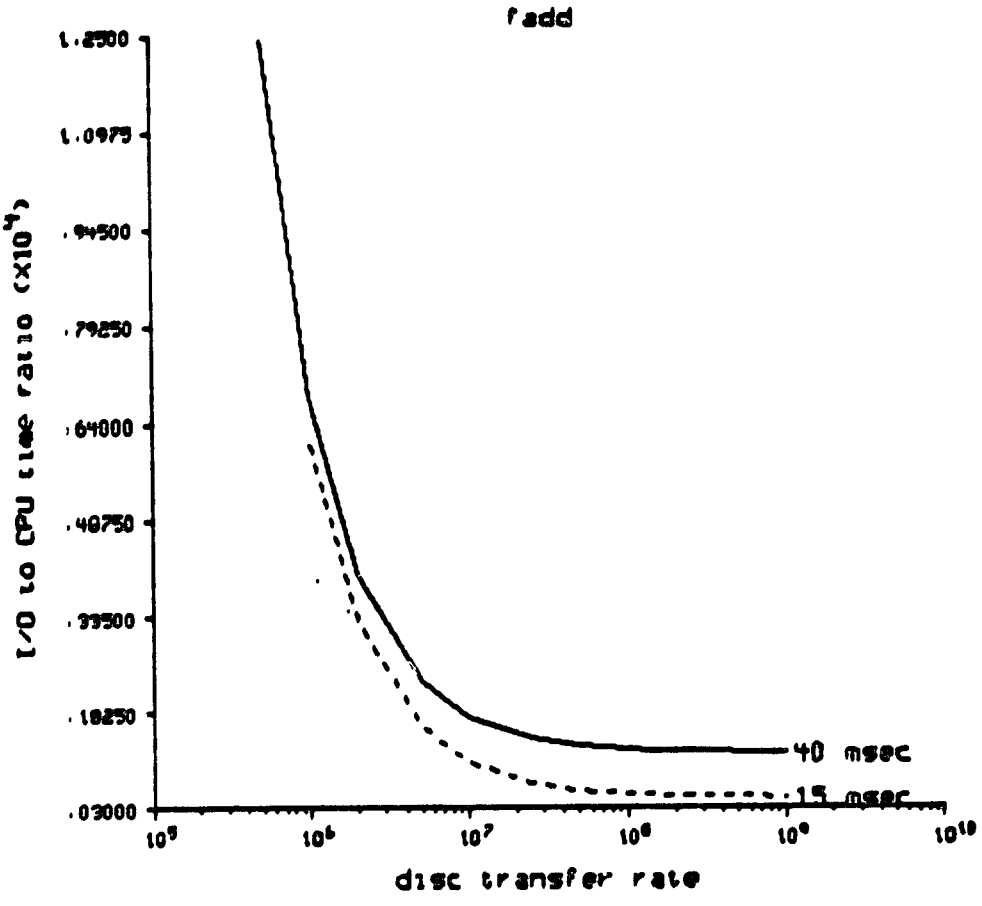


Figure 2: I/O-CPU Time Ratio: floating-point addition

longer than the computation. If a secondary memory is involved, its latency must be very low and its transfer rate very high. Figure 3 shows the dependence of X upon the disc transfer rate for two values of L_d.

The gap between the input-output time and the computation time is narrower than in either of the two cases above; for L_d=15ms and R=10^8Mb/s the ratio is

$$X = 21.83 + (1.776\times10^4)(1.5\times10^{-2}) + \frac{3.49\times10^9}{10^8} = 323.13$$

5.3.2.4 I/O-CPU Time Ratio: sequence of operations

In the previous three cases, X has been computed assuming that each data item (128×128 array) is transferred individually and used only once. If instead the data is ''spooled'' - transferred in blocks - the access latency will be ''spread out'' across many more elements. For instance, if a block of N 128×128 arrays of floating-point numbers is transferred in one operation, X can be computed as

$$X = \frac{(2.91\times10^8)(N\times32)(10^{-7})}{128} + \frac{(2.91\times10^8)L_d}{16384} + \frac{(2.91\times10^8)(N\times32)}{8R}$$

Note that now X is the number of operations that must be performed on the set of N arrays; the number of operations per transfer is X/3N. Figure 4 shows this value for transfers with different block sizes (i.e. values of N).

Problems which are best suited to a parallel matrix processor are usually very computationally intensive. It is an

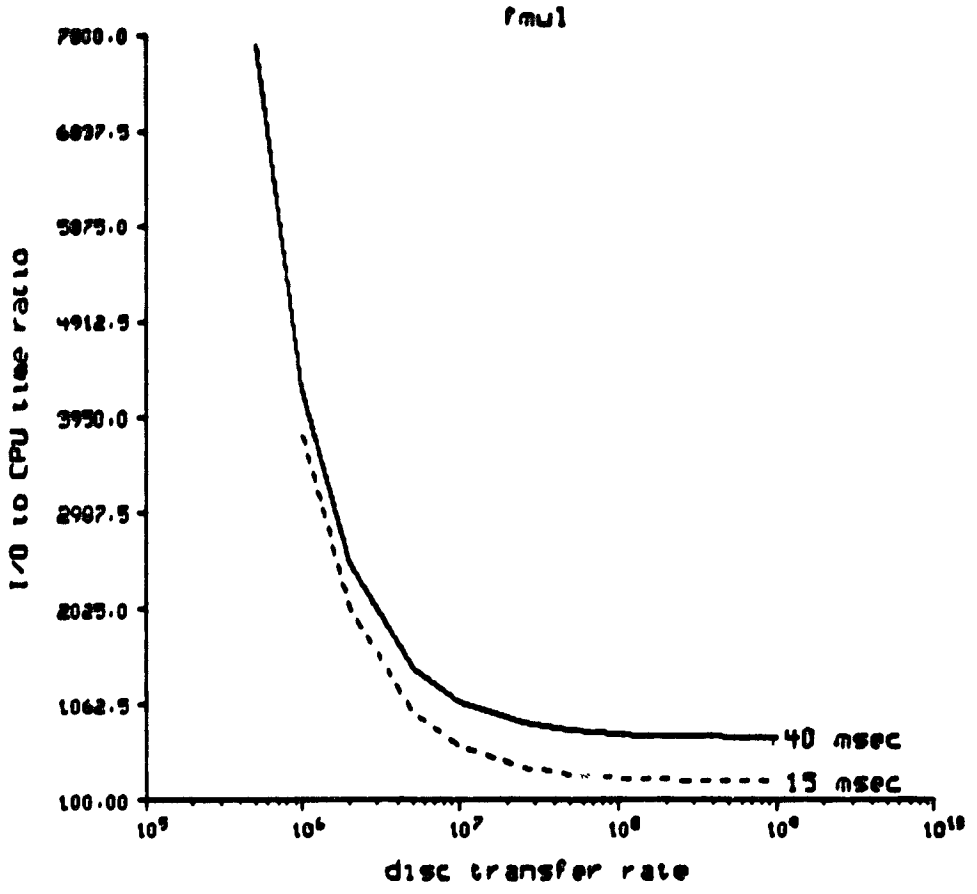


Figure 3: I/O-CPU Time Ratio: floating-point multiplication

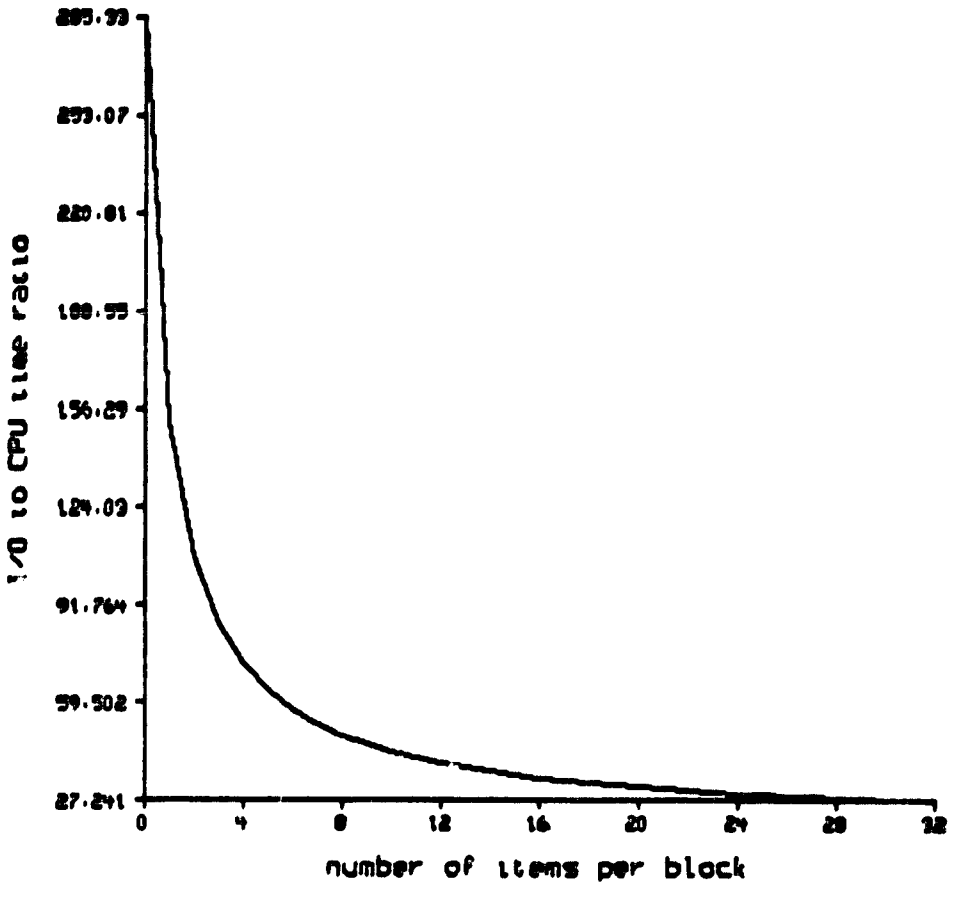


Figure 4: Dependence of I/O-CPU time ratio on block size

extreme case to input two operands, perform an operation, and output the results. A far more common occurrance is to input the operands, perform many operations upon them, and then output the results. As the previous sections have described, X indicates the ratio of the input-output time to the computation time. An equivalent way of expressing this is that if N operations are performed per transfer, N<X, and input-output is completely overlapped with processing, the PE array utilitation will be N/X. (If N>X then the PE array utilization will be 1.) Figures 5 and 6 illustrate the effect of the disc transfer rate (with 15 millisecond access latency) and the number of operations performed per transfer upon the PE array utilization, for a sequence of floating-point multiplications. Figure 5 assumes that each 32-bit floating-point operand is transferred individually; figure 6 assumes that the data is spooled with a block size of 512 bit planes (sixteen 32-bit operands are transferred to/from the disc at once). If the disc is sufficiently fast, the operands are spooled (transferred in blocks), and each value is used in many operations, a totally-overlapped input-output system can keep the PE array busy 100% of the time.

5.3.2.5 I/O-CPU Time Ratios: Conclusions

It is not particularly surprising that the MPP can process data much faster than it can perform input and output. However, the discrepancy between the input-output speed and the processing

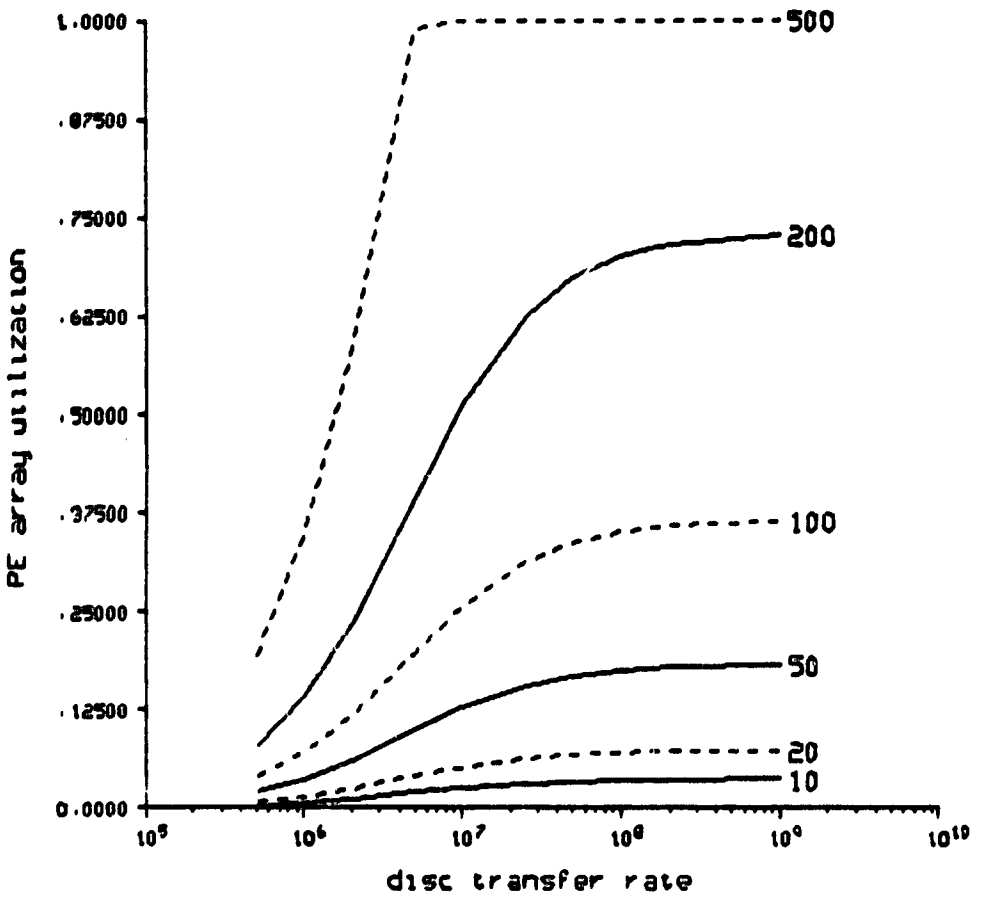


Figure 5: Dependence of PE utilization on transfer rate: unspooled

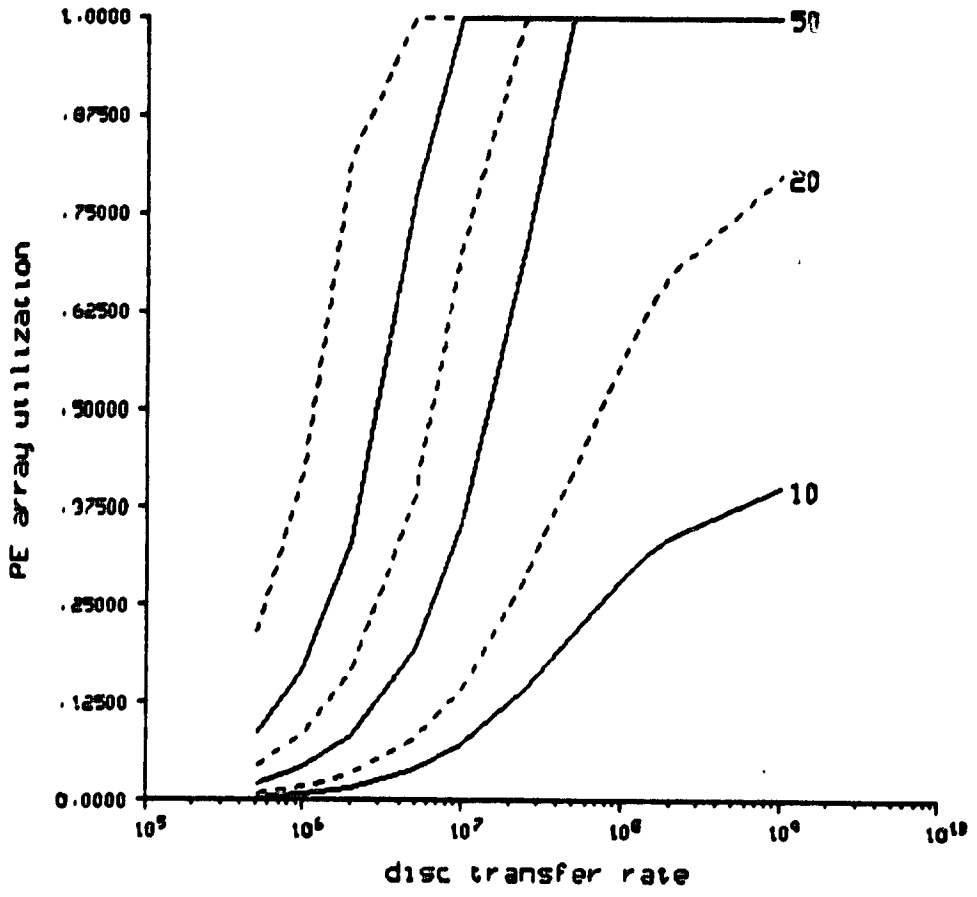


Figure 6: Dependence of PE utilization on transfer rate: spooled

speed can be very large. This suggests several things.

First, it is absolutely essential that data transfers be kept to a minimum. Data should be read in and written out only once if at all possible. It is far more important to minimize the number of transfers than to achieve maximum overlap between input-output and processing, for a system which performs the minimal amount of input-output but does not overlap that input-output with computations is at worst one-half as fast as the optimum system.

Second, a high-speed secondary memory is absolutely essential. The very optimistic figures for L_d and R still produced a high ratio of input-output time to CPU time. As delivered, the MPP will instead have a six megabaud link to the host processor. Unfortunately, although this link itself is slow relative to the MPP processor speeds, the limiting factor in this system will be the memory systems which are attached to the host. Some standard disc subsystems for the VAX are listed in Table 1 along with their average access times and transfer rates[2]. Considering the analysis of the preceeding section, discs with such high access times and low transfer rates (relative to MPP processing times) will severely limit the performance of the system.

Table 1: Access Times and Transfer Rates for DEC Discs

disc	L_d access time (milliseconds)	R transfer rate (kilobytes/second)
RM02	42.5	806
RM03	38.3	1200
RM80	33.3	1200
RP06	38.3	806
RM05	38.3	1200
RP07	31.3	1300
RP07-D	31.3	2200

Third, transfers should be performed between the staging buffer and the PE memories whenever possible. The gap between the input-output time and the processing time is relatively narrow when only the staging buffer is involved. When sequences of complicated operations (e.g. intensive floating-point calculations) are performed it will be possible to significantly overlap transfers between the PE array and the staging buffer.

Finally, when it is necessary to transfer data to the secondary memory it should be transferred in relatively large blocks. The dominating factor in the input-output time to the disc is the access latency; hence, it is desirable to transfer as much as possible when input-output must be performed.

5.3.3 Implementation Alternatives

There are two possible implementation schemes for data migration on the MPP. The transfer of data between memories may be handled by a memory-management system (and hence be transparent to the programmer) or it may be directly programmer-

specified.

5.3.3.1 Automatic Data Migration

Chapter 3 described the implementation of Parallel Pascal through the intermediate language Parallel P-code. One of the significant characteristics of Parallel P-code is its stack orientation. The amount of memory (excluding dynamically allocated memory, which is runtime-dependent) which the main program and each function or procedure require (per call) can be determined by the code generator at compile-time. Given an unbounded memory size, the memory address of the next temporary location can be easily determined (it is the address following the top-of-stack).

Unfortunately, the available memory on the MPP is not unbounded; on the contrary, it is very small. If an automatic memory management scheme is to be used, some locations in the main memory must be shared by several different data items. Hence, the main memory, staging area, and secondary memory form a three-level memory hierarchy.

Conventional machines often utilize memory hierarchies at two levels. The first is the addition of a hardware cache memory to supplement the bulk main memory of the machine. This is usually implemented solely in the hardware of the machine. The second level is the implementation of ``virtual memory'', which migrates data from main memory to secondary (disc) memory. This allows a program to use a large (virtual) address space without

requiring that all of that address space be physically resident in main memory at all times. Virtual memory is typically performed by software, with appropriate hardware support.

The MPP memory hierarchy does not greatly resemble a cache memory system. Cache memories are usually an order of magnitude faster than the main memory of the computer and a few orders of magnitude smaller. For example, in the PDP-11/70 the cache memory has an access time of 0.3μs and a capacity of 2 kilobytes, while the (magnetic core) main memory has an access time of 1.32μs and a typical capacity of 512-1024 kilobytes. In the MPP, on the other hand, the time to access data from the staging buffer is approximately two orders of magnitude greater (it requires 129B cycles rather than B cycles to fetch B bits), while in the delivered version the main store is equal in size to the staging buffer. (Even with a full complement of memory, the staging buffer will only be 32 times larger than the main memory.) Therefore, consideration of the MPP memory hierarchy as a cache memory seems ill-advised.

The MPP memory hierarchy also does not greatly resemble a virtual memory hierarchy. Conventional machines typically operate in a multiprogrammed environment (i.e. several programs running concurrently). These programs compete for (share) the main memory and other machine resources. When an executing program attempts to reference data which is not resident in main memory a page fault occurs. The operating system transfers the desired data from the secondary memory to the main memory, and

when it is accessible the program is restarted. While the transfer is taking place the program is suspended and another program is allowed to run. In the ideal case, the processor is always busy even though one or more programs is currently blocked. The MPP, however, is not multiprogrammed. When the program is blocked awaiting input (or output) the array of processing elements is idled. As section 5.3.2 noted, input-output (especially to a secondary memory) is very expensive.

Without actual runtime experience it is difficult to predict the type of automatic memory management system which would be most effective. Without such experience it seems advisable to consider some qualities that such an implementation might possess, instead of attempting to fully define the implementation.

First, because transfers between the staging area and main memory are the least expensive (for sequences of complex operations such as floating-point arithmetic it will be possible to overlap most of the input-output with other computations) the staging buffer should be used to hold variables and temporary data which will be needed again, and the secondary memory should be used only for input of the original problem and output of the results (or, if necessary, overflow from the staging area).

Second, all transfers between the staging area and the secondary memory should consist of blocks of data. If necessary, data may be loading into the staging area before it is needed in

order to avoid later disc references (with their long access latencies). This is analogous to demand prepaging in virtual memory systems[3]. Some form of data restructuring may be used to improve the clustering of commonly-used locations into contiguous locations[4].

Finally, the stack orientation of Parallel P-code can be used to advantage. When a function or procedure is called recursively the data locations corresponding to the previous activation of that routine become inaccessible (until the recursively-called routine returns). Temporary locations at any outer lexical level are also inaccessible until control returns to the routine which calculated them. If data migration is necessary, these locations could be used, especially those at the outermost lexical levels (for which the next reference will be a relatively long time in the future). If memory planes are shared by several small arrays (those with dimensions less than the hardware array size), this method can still be used provided that memory planes are shared only by variables (or temporary data) within the same procedure activation.

5.3.3.2 Programmer-Directed Data Migration

The alternative to an automatic memory management system is a programmer-directed system. Such a system requires the programmer to be concerned with the implementation details; it is therefore less portable and somewhat more difficult to use. However, it has the potential for higher system performance since

no potentially inefficient data transfers are performed ``behind the programmer's back.'' The initial implementation of Parallel Pascal on the MPP will use this scheme, as described in section 2.3 of this report.

5.4 References

1. Kenneth E. Batcher, "Architecture of a Massively Parallel Processor," Proceedings, International Symposium on Computer Architecture (1980).

2. Peripherals Handbook, Digital Equipment Corporation, Maynard, Mass. (1981-82).

3. Kishor S. Trivedi, "On the Paging Performance of Array Algorithms," IEEE Transactions on Computers Vol. C-26(10), pp.938-947 (October 1977).

4. Jehan-Francois Paris, "Application of Restructuring Techniques to the Optimization of Program Behavior in Virtual Memory Systems," UCB/ERL M81/44, College of Engineering, University of California, Berkeley, CA (18 May 1981).

6: FAULT TOLERANCE IN HIGHLY PARALLEL MESH CONNECTED PROCESSORS

6.1 Introduction

The mesh interconnection scheme has been used on several large scale SIMD parallel processors. This scheme involves organizing the processing elements (PE's) into a two dimensional matrix such that each PE has data interconnections with its adjacent neighbors. In a typical organization a PE has connections to 4 near neighbors in the cardinal directions N, S, E and W. In a single instruction data may be shifted in a single specified direction between all adjacent PE's. That is, a distributed matrix of data may be shifted one mesh position in parallel. The main advantage of the mesh scheme is its simplicity and suitability for a large class of scientific applications. Data interconnections only occur between adjacent PE's this means that they may be kept very short and laid out on a single interconnection plane. The usual disadvantage with this scheme is that data transfers to distant PE's require a large amount of time since the data can only cross between adjacent mesh nodes with each clock cycle. However, there is a large class of problems including physical system modeling using partial differential equations and image processing in which the data needed by a PE is located in its local mesh area and the mesh interconnection scheme is very efficient.

One potential problem with the mesh scheme is that the failure of any node in the mesh renders the whole parallel processor inoperable. Current LSI processor designs involve a mesh with more than 10,000 nodes; with VLSI technology systems having 1,000,000 nodes and more may be anticipated. For some applications, for example real-time image processing in a remote inaccessable robot system, some fault tolerance is essential.

6.2 Mesh Connected Parallel Processors

An important large scale mesh parallel processor is the Illiac IV [1] developed in the late 1960's. It consists of an 8 x 8 mesh connected set of 64 PEs; each PE having an ALU with a 64-bit-wide datapath and floating point capabilities. This architecture is well suited to applications such as partial differential equations. The implementation of Illiac IV was hampered by the technology of the time. Hardware failures were anticipated to occur every few hours. The PE's were regularly subjected to an extensive library of automatic tests and were replaced manually if any faults were detected.

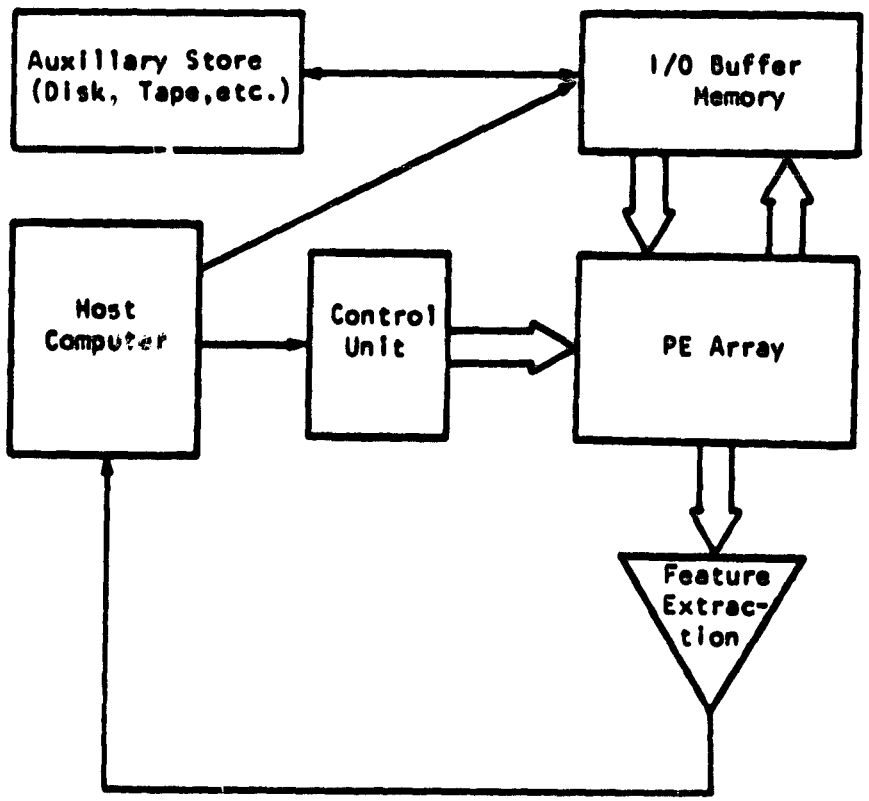
A more recent design, based on LSI technology, is the Massively Parallel Processor (MPP) [2,3] which is currently being developed for NASA by Goodyear Aerospace and should be constructed by late 1981. The MPP consists of a 128 x 128 mesh of 16,384 PE's; each of which has a 1-bit-wide data path and can achieve floating point operations through bit-serial algorithms. This architecture is designed for image processing applications where a single image may be as large as a 6000 x 6000 matrix. The MPP processes such an image as a sequence of 128 x 128 subimages. The MPP involves parity checks on each 8 bits of the PE local memories and has a redundant column of PE's which may be switched in by the host computer to replace a faulty column. With this fault tolerance the MPP is expected to run for several hundred hours before requiring manual intervention.

The fault tolerance concepts in this paper will be considered with respect to a bit-serial PE array scheme or Binary Array Processor (BAP) [4] such as the MPP. These concepts may be extended to word parallel designs such as the Illiac IV type architecture. However, with the constraints that the matrix to be processed is larger than the PE array and that the algorithms to be implemented are well formed for the mesh organization, the bit-serial approach has significant advantages over the word parallel approach for equivalent amounts of hardware [5].

A general block diagram for a large scale BAP is shown in Fig. 1. Data processing is achieved with the array of PE's. Data is input to and output from the PE array via the I/O buffer memory which communicates the data to peripherals and bulk auxiliary storage devices. Instructions to the PE array are issued by a single high-speed microprogrammed control unit. The whole system synchronization is maintained by a conventional host computer which issues macro instructions to the control unit. Some feature information may be extracted from the PE array by the global information extraction mechanism.

A typical organization for an MPP-like PE is shown in Fig. 2. Data from adjacent near neighbors is selected by the NN multiplexor. The control lines and local memory address lines are broadcast to all PE's in the array. The OR bus is a line from all PE's to the control unit which has a one value if any PE outputs a one. The I register is used for data I/O; it receives data from the I register of the adjacent PE to the left and transmits data to the PE on the right.

The I/O buffer memory is vital part of the BAP system, it is responsible for making reformated data available to the PE array. With the MPP a data matrix is input to the array as a set of bit-planes. Each bit plane is input along one edge



Binary Array Processor System

Fig. 1

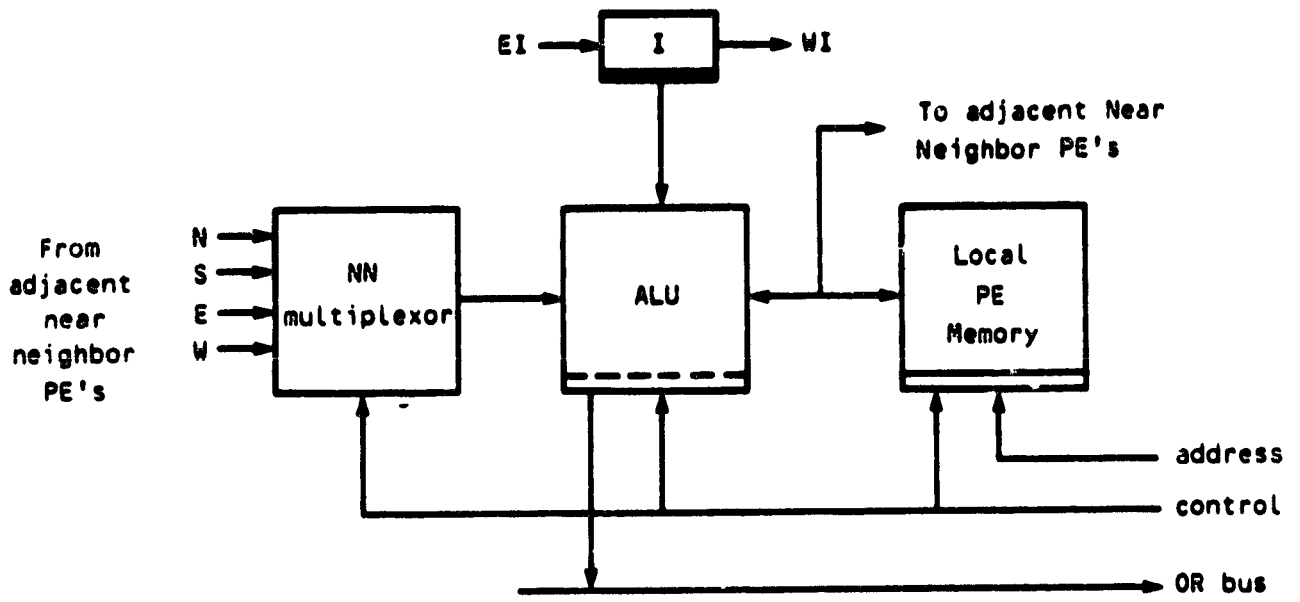


Fig. 2. A typical BAP PE organization.

of the PE array; one column with each clock cycle. Each row of the array acts as a shift register. When the complete bit plane has been input it is stored in the PE local memories in one clock cycle. Fault tolerance in the I/O Buffer can be achieved with the single error correction-double error detection (SECDED) schemes common in many recent large memory systems. For the PE array the I/O mechanism is like a one dimensional mesh connection; and it will not be treated as a separate issue for fault tolerance. For a BAP system the I/O could be achieved by the mesh interconnection hardware; alternatively a separate but basically similar I/O hardware as shown in Fig. 2 could be used, if necessary, to avoid blocking.

The global feature extraction mechanism on the MPP is an OR function over all PE elements, which outputs a 1 if any PE has a 1. If we have an error detection mechanism then a similar global OR function would be needed to report an error to the host processor. Once again these two functions will be considered to be implemented with the same hardware in this paper; such a scheme is used with the MPP [3]. It has been suggested that a more powerful feature extraction mechanism, such as counting the number of bits set in a bit plane may be cost effective for future BAP systems [6].

The MPP PE array is constructed with two LSI chips, A PE chip and a memory chip. The PE chip contains 8 PE's (without local memories) in a 2 x 4 array. Each PE chip has connections to 8 1-bit memories for the PE's and an additional 1-bit memory for a parity check of the other 8. The total PE array consists of 33 4-PE wide columns; each column consisting of 128 PE chips. A PE chip has a control input which, when activated, disables the chip by connecting corresponding East West pin data lines together. In this way any one of the 33 columns may be disabled to achieved on operational 128 x 128 PE array. When an fault is detected the faulty column is disabled and the redundant column is used to replace it.

Faults will be considered here to be of two basic types - local and module. A local fault may typically be a broken data line or a faulty memory bit whereas a module fault implies the complete failure of a module, such as a chip, which may result in a set of related PE's being made inoperable.

Since we are dealing with functionally very complex chips the probability of a local fault may be expected to be significantly higher than a module fault. Therefore the main effort of the work here is concerned with local faults as they

are much simpler and cheaper to deal with. However, any practical very large scale mesh connected processor also needs some fault tolerance at the module level.

For the MPP, the redundant column scheme is effective for any single memory chip failure. It is also effective for most PE local failures, e.g. if a data line breaks in a PE. Therefore the most probable fault causes have been covered. However, if a catastrophic failure occurs to a PE chip, (module) then the whole PE array may become inoperable since it is necessary for data to flow through a disabled chip.

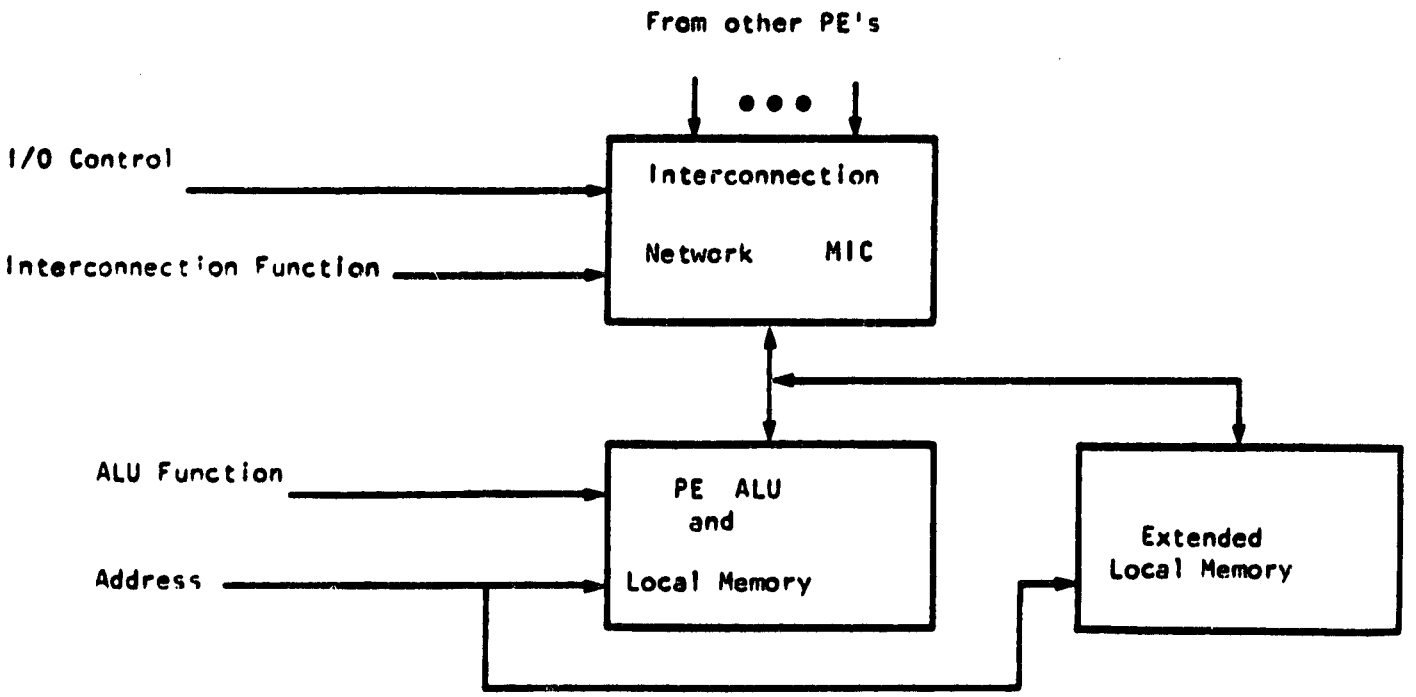
6.3 A VLSI PE Array Organization

For a VLSI system design there are two fundamental chip size limitations (1) the number of devices which can be put onto a chip and (2) the number of pin connections which may be made to the chip. A usual characteristic of a VLSI design is modularity, i.e. a chip consists of a very large number of identical modules, which is important to minimize the development cost. Finally, with very large functionally complex chips fault tolerance may be effective to significantly increase the production yield and the fault free lifetime of the chip.

A possible chip organization for a very large VLSI PE array is shown in Fig. 3. Three different VLSI chip types are involved: a PE ALU chip, a local memory chip and a PE mesh interconnection chip (MIC).

The PE ALU chip consists of a set of PE ALU's, each having a limited amount of local memory. These ALU's share common ALU-function and address lines but do not have any data interconnections between each other. The data access to a PE is achieved by a single pin on the chip which is connected to a bidirectional bus line. The design of an effective PE with this input/output constraint is described in [5]. With this design optimal bit-serial processing times for addition, multiplication and logical operations can be achieved. The limited size on-chip local memory may be used for table-look-up applications since it may be addressed by an ALU register (unlike the external local memory) or for a cache memory.

The PE ALU chip will be a functionally very dense chip and will contain as much logic as the VLSI technology will allow. There are no pin connection problems since only one pin is required for each additional PE.



VLSI PE Organization

Fig. 3

The external local memory chip will provide the main local data storage for a PE. With the amount of single chip storage which is becoming available with emerging memory technology, it is possible that one VLSI memory chip could contain adequate storage for one or even several PE's. The 1-bit wide external PE memory is connected to the single-bit PE data bus. Once again the limitation with the memory chip is caused by the functional complexity achieveable with the VLSI technology; there is no pin connection problem.

The interconnection chip realizes the mesh interconnections between the PE's and also contains an input/output mechanism for data I/O to the PE array. Unlike the previous chips this chip is functionally very simple and the size of the mesh which can be contained on a chip is limited by the maximum possible number of pin connections. Each mesh node requires one pin connection to its PE data bus and also, for a m x n mesh, 2(m+ n) additional data interconnections are needed to adjacent MIC's.

All the additional logic to achieve error reconfigurability for the PE array is located in the MIC's.

6.4 PE Fault Tolerance

Both the ALU and external memory chips are functionally very complex and therefore more likely to fail than an MIC. In this section we consider how to reconfigure the array if a single PE-external memory combination fails. This reconfigurability is achieved by modifying the MIC so that it has access to spare PE's which may be switched in to replace the faulty PE.

A basic non-fault-tolerant MIC organization for a 2x2 mesh subsection of a PE array is shown in Fig. 4. This chip has a total of 12 data pin connections; 1 to each of the 4 PE's and 8 to adjacent neighbor MIC's. In general a m x n mesh MIC would require mn+ 2m+ 2n data pin connections.

The basic logic device which the design of the MIC will be based on is the selector which is illustrated in Fig. 5(a). A selector has a set of control inputs, C, which specify by a binary code which of the X data items is to be connected to the Y data line. Once connected, data may flow in either direction from X to Y or Y to X. With some logic technologies an additional control input may be needed to specify the data flow direction. However, with designs considered here, the direction information is always locally available therefore this additional control line is no problem.

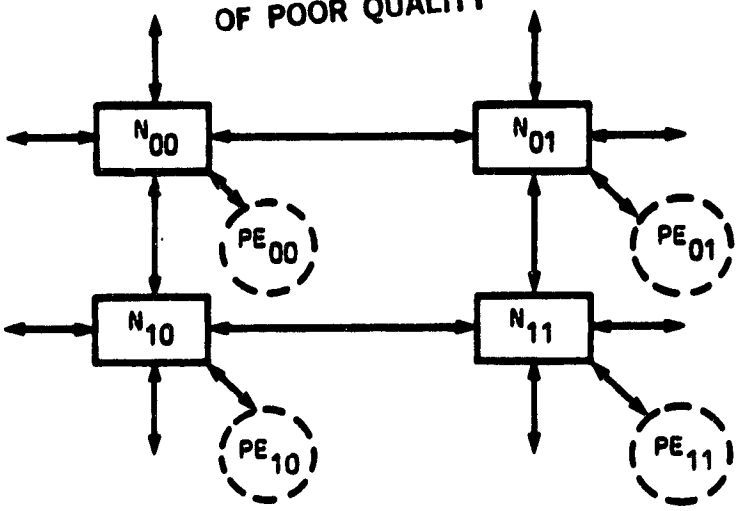


Fig. 4. A simple 2 x 2 mesh interconnect chip organization.

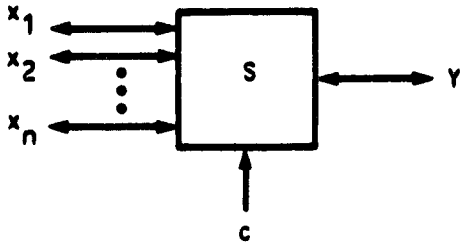


Fig. 5(a). A selector switch.

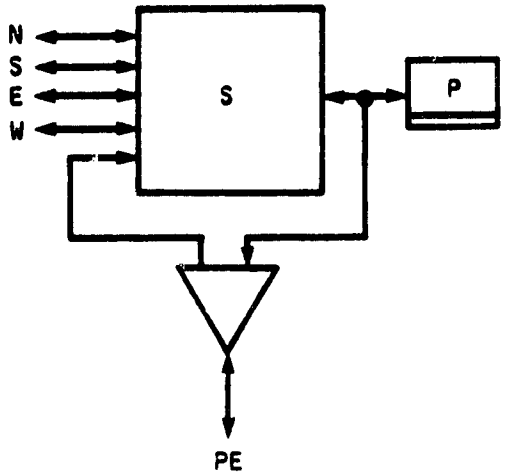


Fig. 5(b). A simple MIC mesh node organization.

Each mesh node of the simple MIC shown in Fig. 4 may be implemented with a 5-way selector and a 1-bit register as shown in Fig. 5(b). Two clock cycles are required, with this design, to transfer data between adjacent PE's. In the first clock cycle the data is output from the PE and loaded into its mesh node P register. Then, in the second clock cycle, the PE reads the value of an adjacent PE's P register. Data may be transferred between more distant PE's by shifting it through a connected sequence of P registers. In general, a data transfer through a path of K stages requires K+ 1 clock cycles.

To achieve fault tolerance to a single PE failure we first consider adding a spare PE to each MIC group. A possible organization for such a reconfigurable MIC is shown in Fig. 6. Each mesh node may be connected to one of two PE's. If any PE fails each mesh node can be connected to a unique, operational PE.

The details of the modified mesh node design are shown in Fig. 7. A Q register and a 2-way selector have been added to each node. The value of the Q register specifies which of the two possible PE's the mesh node is connected to. When a faulty PE is identified the host computer generates a bit mask which is distributed to the P registers; then the Q registers are loaded from the P registers to isolate the faulty PE. The task which was in progress when the faulty PE was detected must be reloaded or restarted.

The above MIC modifications require only two new pin connection to the MIC. One is the load control and the other is the data connection to the extra PE. One extra PE must be available to each MIC; however, it is possible for MIC's to share a PE as indicated by the broken lines in Fig. 6. In this case only one extra PE for a group of MIC's is needed.

The above technique is easily extended if protection against more than one faulty PE for each MIC (or group of MIC's) is required. For example, protection against any two faulty PE's could be achieved by connecting two extra PE's to the MIC as shown in Fig. 8. The PE selector at each mesh node must select between three PE's, and the Q register must be extended to contain two bits of information. In the general case, fault reconfiguration for the up to K faulty PE's requires K extra PE's; each MIC requires K+ 1 more pin connections than for no protection. Each mesh node in the MIC must contain a K+ 1 way PE selector and a Q register large enough to address it.

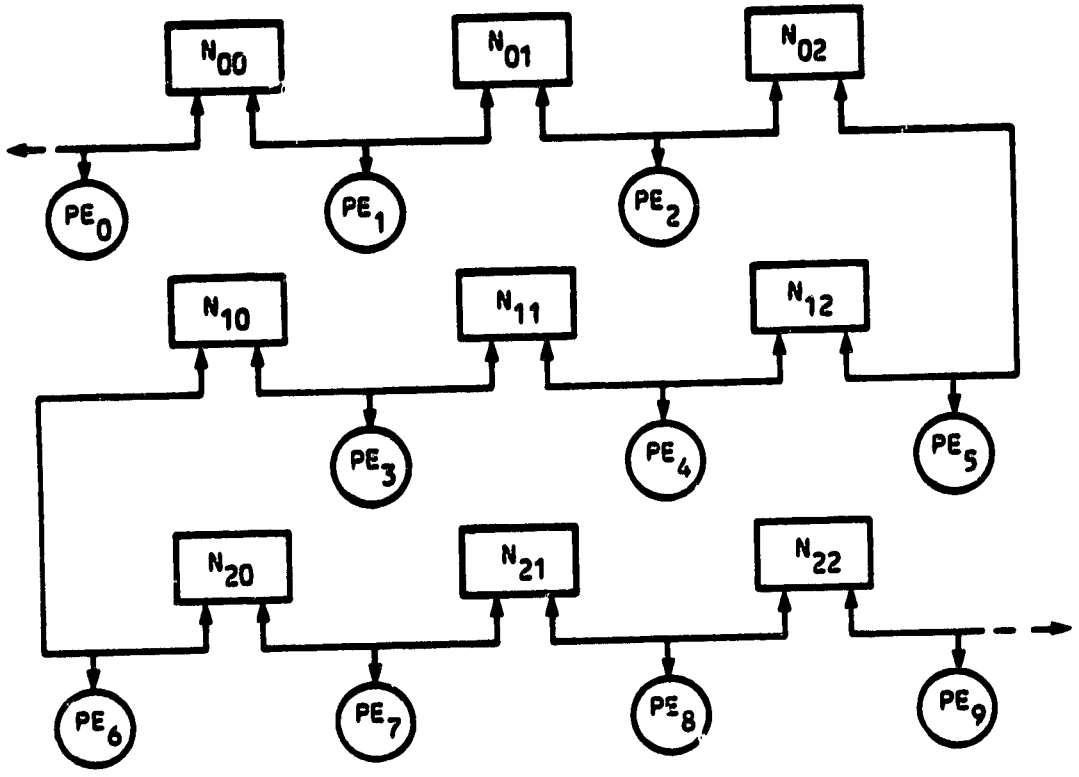


Fig. 6. A 3 x 3 matrix of interconnection nodes connected to 10 PE's.

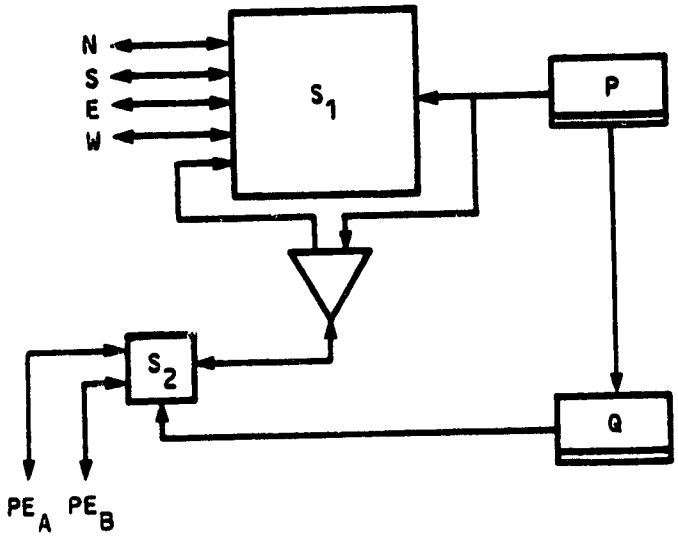


Fig. 7. A mesh node with PE fault tolerance.

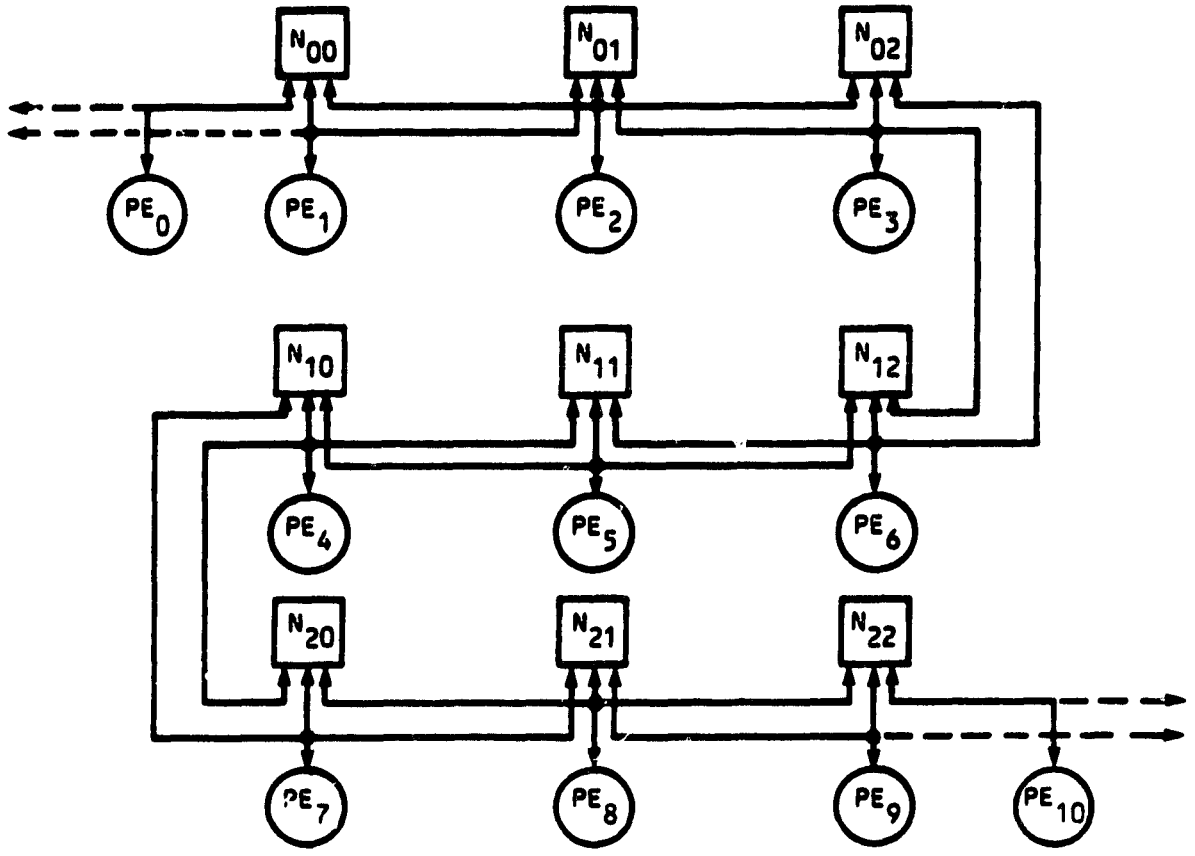


Fig. 8. Organization for fault tolerance to any two faulty PE's.

6.5 MIC Mesh Node Fault Tolerance

Once the system may be reconfigured for any faulty PE, the next problem area is the very large mesh interconnection network itself. Fault tolerance is considered here for the failure of any mesh node or data interconnection in the interconnection network.

Mesh node fault tolerance on the MIC can be achieved using a similar scheme to the MPP global fault tolerance. That is, have a spare column of mesh nodes which may be utilized when a faulty mesh node is detected. One way in which a spare column of mesh nodes may be incorporated within an an MIC is illustrated for a 2x2 mesh MIC in Fig. 9. In the general case with n+1 columns the configuration is specified by a register (not shown in Fig. 9) having two bits for each column. A possible organization of a mesh node for this organization is illustrated in Fig. 10. The two bits from the reconfiguration register are represented by RL and RR. When RL is set it specifies that the left (W) input to the node not be connected to the adjacent column node but to the next node to that, i.e. to skip the node to the left; RR specifies which column node is connected to the (E) input in a similar way. A bit pattern is loaded into the reconfiguration register such that one column is skipped. It does not matter what values a disabled faulty node may have on its interconnection lines since these lines are never used by the other nodes. For the rest of this paper the configuration in Fig. 10 will be considered to be implemented by a single 7-way selector.

Any external data interconnections pin may be connected to one of two mesh nodes; therefore it is necessary to have a 2-way selector associated with each node as shown in Fig. 9. The control for these selectors is derived from the reconfiguration register contents.

The simple organization shown in Fig. 9 can reconfigure for any faulty mesh node however, there is no fault tolerance from either a data pin connection failure or a data pin selector failure. Fault reconfigurability for such failures may be achieved by adding spare pin connections and selectors, one for each of the four directions of data connection as shown in Fig. 11. Only the connections to pin selectors are shown, the interconnections between mesh nodes is similar to Fig. 9. This organization assumes that the MIC's are themselves connected in a matrix. Now if any pin selector or connector fails the two remaining pin connections may be used. The MIC connected to this chip must also use the same data connections, therefore, we have fault tolerance to any single selector or data connection

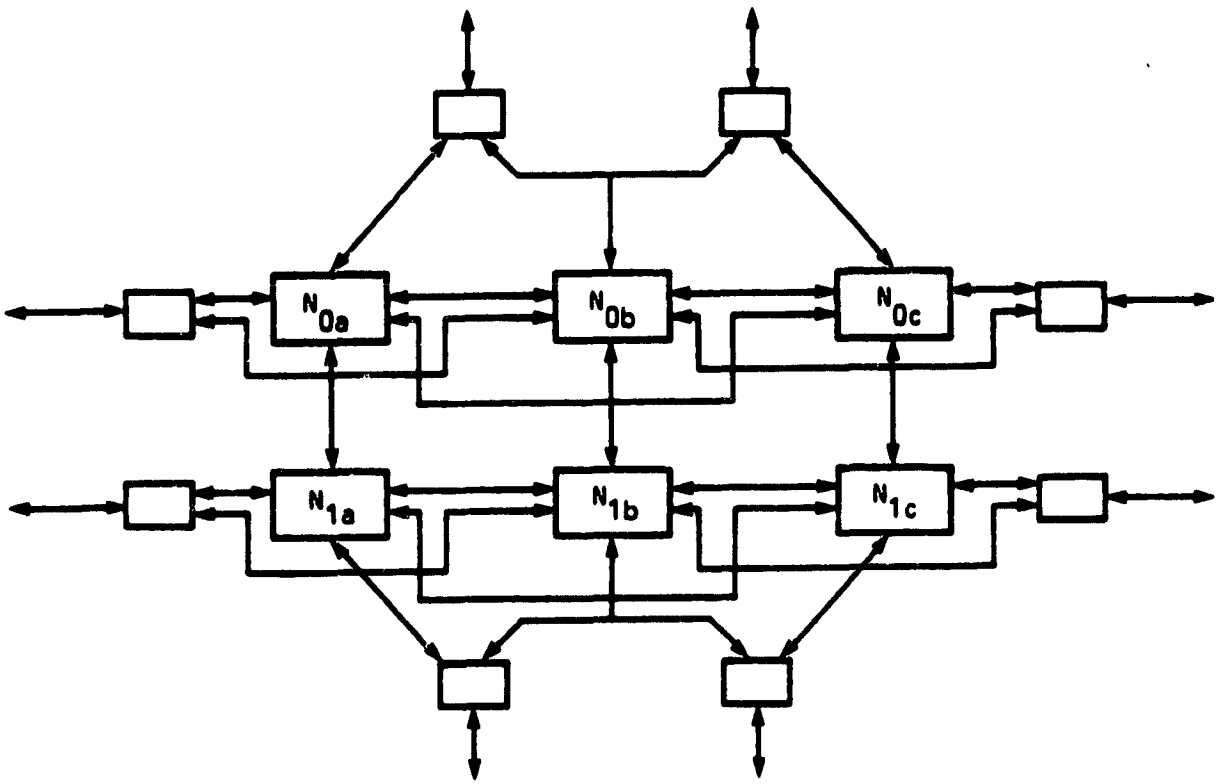


Fig. 9. MIC organization for fault tolerance to any single mesh node failure.

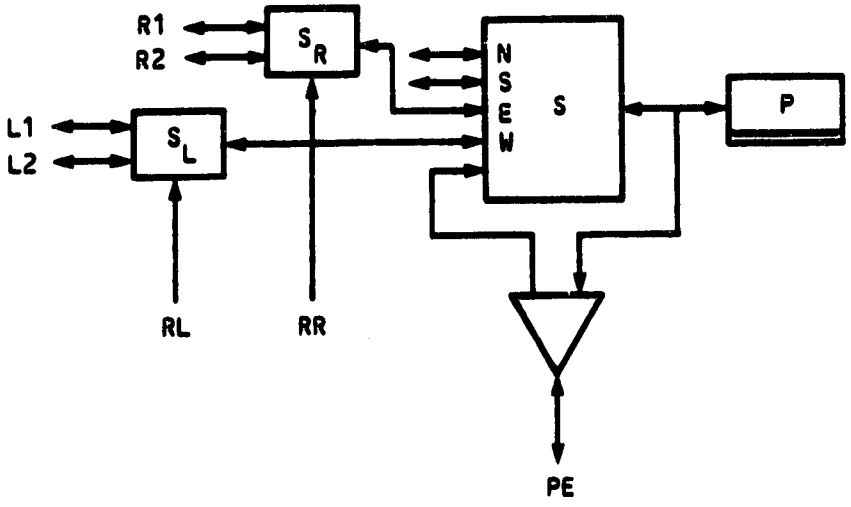


Fig. 10. A column reconfigurable mesh node.

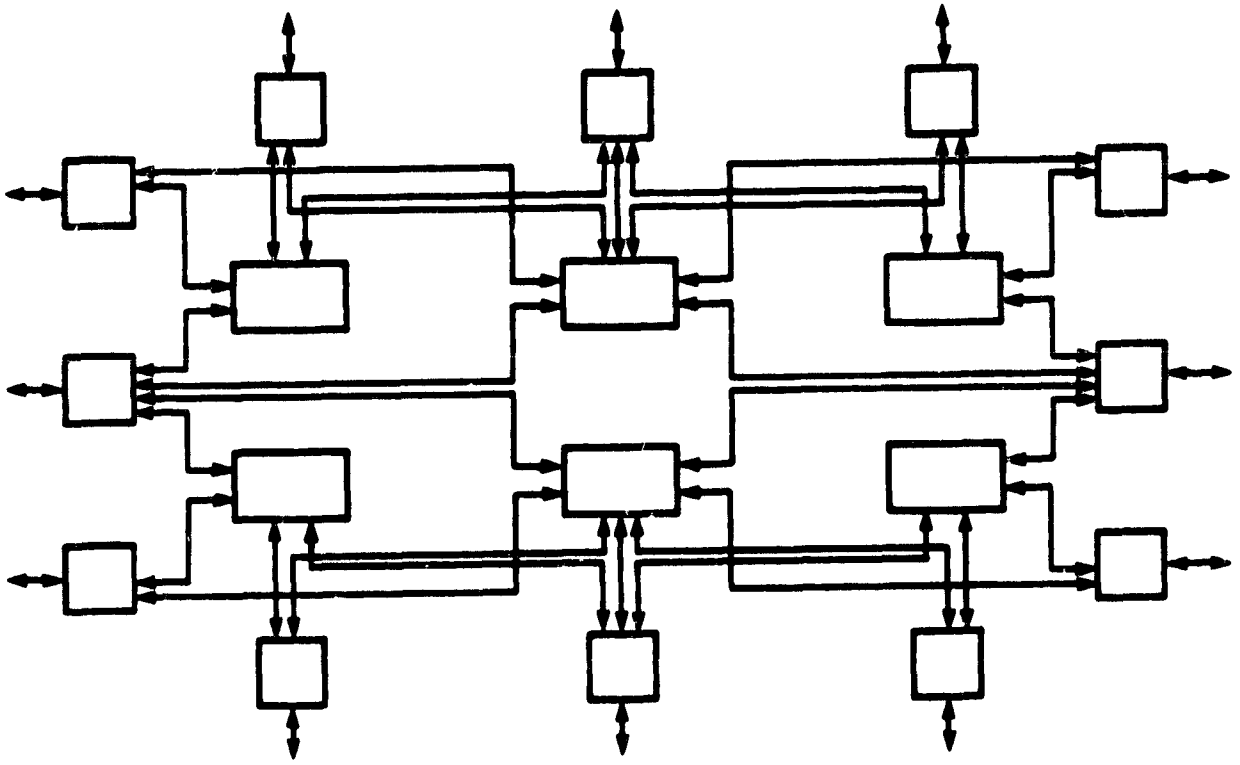


Fig. 11. Pin connector and pin selector organization for complete data line fault tolerance.

failure between the interface of two MIC chips. In the general case, this fault tolerance requires four extra pin connections and pin selectors. Furthermore, except for selectors at the end of the rows or columns, column pin selectors are 3-way and row pin selectors are 4-way.

To complete the reconfigurable mesh node design the PE's must be connected to the enabled mesh nodes. One way of doing this is illustrated in Fig. 12. For a 2x2 active mesh there is one extra column of mesh nodes and one extra PE. Since any mesh node may fail the node-PE selector is associated with the PE rather than the mesh node in contrast to the PE-only fault tolerance shown in Fig. 6. In the general case, each PE must be connected to a mesh node by either a 2-way or a 3-way selector to the mesh nodes.

Finally, we note that there is a simple extension to this scheme to achieve reconfiguration for any two faulty mesh nodes. This may be done by having either two extra columns on one extra row and one extra column. In either case all the mesh nodes require an additional two data connections. In general a second spare column will be cheaper than an additional row. For example for an n x n MIC a spare row requires n+ 1 mesh nodes whereas a spare column only requires n mesh nodes.

6.6 Module Fault Tolerance

Fault tolerance to catastrophic chip failures, such as a broken power line or command line may be achieved by organizing the total array into a set of modules. Each module contains a set of related PE's and fault tolerance is achieved by having a spare module available when one fails.

For the discussions in this section an example array design will be considered, however, the techniques discussed here are general in nature and may be applied to system with very different design parameters. The example system could be constructed with present day technology and is for a 1000 x 1000 PE array. The three PE chip types have the following characteristics: each PE chip contains 16 PE's each MIC contains a 4x4 matrix of active mesh nodes and there is a memory chip for every 4 PE's. Fault tolerance will be considered at two module levels (a) the chip level and (b) at the group level where each group consists of a set of chips.

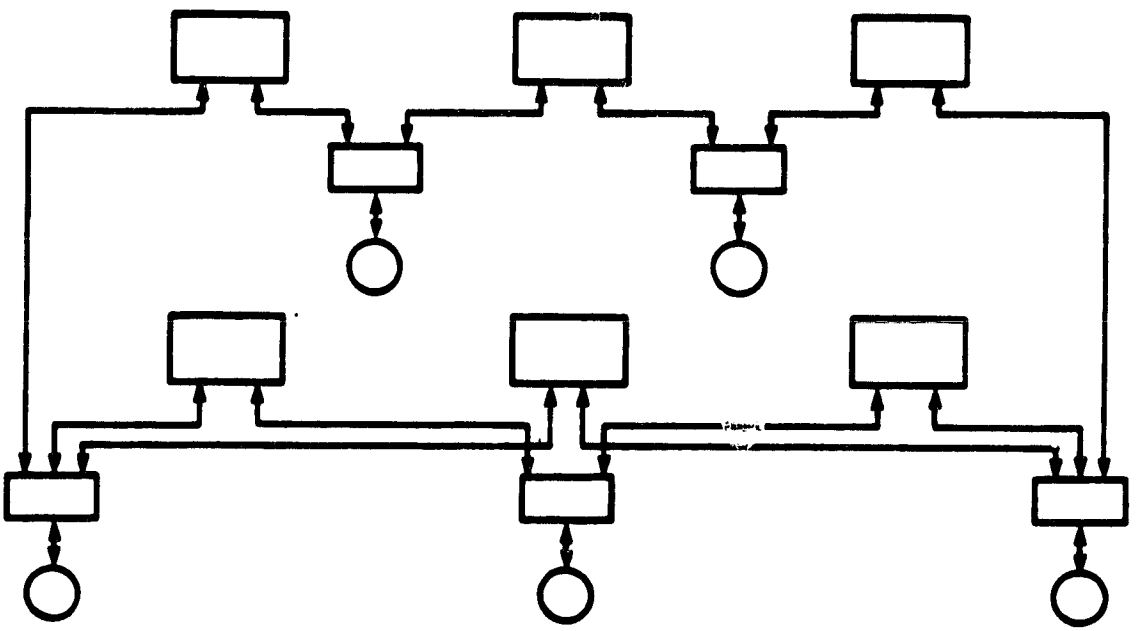


Fig. 12. PE-mesh node interconnections for both PE and mesh node fault tolerance.

6.7 Chip Level Fault Tolerance

A catastrophic fault could occur in any chip, the PE chip is considered first. The smallest possible group size consists of the following, one MIC, one PE chip (plus an extra PE for fault tolerance) and 4 memory chips (plus one bit for fault tolerance). This group, therefore involves a 4x4 matrix of active PE's. If the PE chip fails then it is necessary to replace the whole group.

As an alternative, a larger group may be used involving 256 active PE's in which the PE chips are distributed between the MIC's. This group consists of 16 MIC's, 17 PE chips and 68 memory chips. Each PE chip contributes one PE to every MIC in the group; therefore, since each MIC has fault tolerance to one faculty PE, a single PE chip failure can be handled locally within the group. The cost of PE chip fault tolerance within the group is more complex inter-chip data routing.

The total failure of a memory chip would render 4 PE's inoperable in our example. However, the memory chips may be distributed between MIC's in a similar way to PE chips in order to achieve group local fault tolerance.

If an MIC fails completely then the whole group is rendered inoperable. Therefore we need a mechanism to selectively enable a group in the total array. One approach is to make a group in the form of a column of PE's and have a spare column of PE's in a similar scheme to the MPP. In our example design, a group could be organized as a 64x64 matrix of PE's and 16 groups would constitute one 4 PE-wide column of the PE array; 256 such columns would be required for the complete PE array.

To allow for the disabling of a column each MIC needs to have a spare set of data selectors and data pin connections for all data lines in the E/W directions. This spare set would bypass the adjacent column and link with the spare data connections on the following column. In this way any single column may be completely isolated. Since there are 256 active columns it might be advantageous to have more than one spare columns. Then multiple MIC chip failures could be dealt with as long as they do not occur in adjacent columns.

6.8 Cost of MIC Fault-Tolerance

The cost of implementing the fault tolerance schemes with the MIC has been estimated using three measures (1) the number of selectors, (2) the number of internal data lines and (3) the number of data pin connections. The number of selectors may be considered as a measure of the functional complexity of the chip. No weight is attached to the complexity of each selector and the small amount of control logic is not considered. Although the mesh node selectors are more complex for the fault tolerant design this is balanced by the many additional simplier selectors which are used for PE's and data pins. The number of internal data lines is also a measure of chip complexity since they may consume a large proportion of the chip area. The data pin count gives a good indication of the pin requirements of the MIC since less than 10 additional pins will be required for control and power.

The costs for various fault tolerant MIC configurations are expressed for a m x n mesh design in Table 1. The first row in Table 1 is the cost for the simplest MIC without any fault tolerance. The second row is for single PE fault tolerance as shown in Fig. 7. The third row is for an MIC with complete single mesh node and data interconnection fault tolerance as illustrated in Figs. 9-12. The last two rows include the cost of an extra set of left and right data connections and selectors for group fault tolerance. The first set of figures is for a spare column within the MIC and the second set of figures is for a spare row within the MIC. The spare row concept is slightly cheaper than a spare column for a square mesh i.e. when n = m.

Table 1: MIC cost for an n x m active mesh

Fault Tolerance	Selectors	Internal Data Lines	Data Pin Connections
None	mn	3mn+ m+ n	mn+ 2+ 2n
single PE	2mn	4mn+ m+ n+ 1	mn+ 2m+ 2n+ 1
single mesh node	2mn+ 2m+ 3n+ 5	6mn+ 7m+ 7n-1	mn+ 2m+ 2n+ 5
group (a)	2mn+ 4m+ 3n+ 7	6mn+ 17n+ 7n-9	mn+ 4m+ 2n+ 7
group (b)	2mn+ 5m+ 2n+ 7	6mn+ 15m+ 7n-5	mn+ 4m+ 2n+ 7

These costs are shown graphically in Figs. 13-15. In Fig. 13 the number of selectors for the MIC is shown. While a significant increase in selectors is needed for fault tolerance the chip is still not very complex. For the example system of 16 active nodes only 67 selectors are needed for group fault tolerance; when n=m=10, i.e. 100 active nodes, only 277 selectors are required. In Fig. 14 we see that there is a large increase in internal data lines when mesh fault tolerance is introduced, however the total number of data lines is still quite reasonable. The limiting size design parameter for the MIC is the number of pin connections; in Fig. 15 it is shown that there is only a small increase in total pin connections when fault tolerance is introduced. For the example 16 node MIC 21 data pins are required without fault tolerance and 34 data pins are required for group fault tolerance. When there are 100 active nodes on the MIC the data pin requirements are 140 without fault tolerance and 167 with group fault tolerance.

6.9 Fault Detection

The MPP PE chip for 8 PE's has an additional memory chip for parity information. This mechanism provides good fault detection for the local memory chips. With the VLSI chip organization proposed here a similar mechanism could be implemented. In this case the MIC would monitor all PE local memory reads and writes and store the parity in a separate memory chip. For fault tolerance each MIC may select between two parity memory chips and one spare parity memory chip for each group would be required.

An alternative fault detection scheme is to use additional parity bits with each data operand. The advantage of such a scheme is that data parity may be checked after any data transfers, either I/O or interprocessor, in addition to any memory data transfers. For a bit-serial system this could be implemented with very little hardware in the PE ALU. A single 1-bit parity register and an exclusive-OR gate as shown in Fig. 16(a) is all that is needed for a multibit register ALU. As each operand is read its parity is computed in the T register; then all T registers are output to the global OR function which will report any parity errors back to the host processor. The T register is selected by the local memory address mechanism for setting it to an initial value or reading its contents; therefore, no additional pin connections to the ALU chip are required. This same mechanism is used to generate the parity when a result is stored in local memory or transferred to another PE.

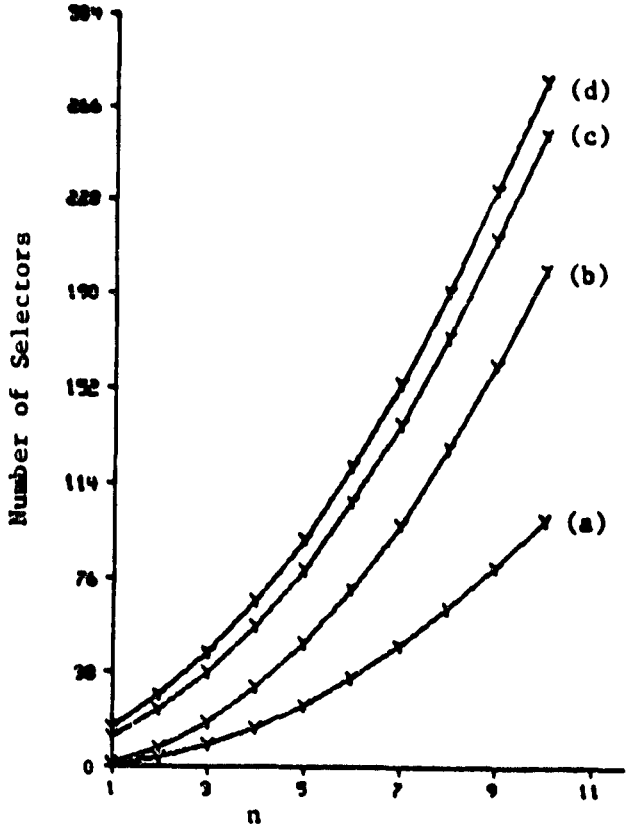


Fig. 13. MIC selector cost for an n x n active mesh. (a) no fault tolerance; (b) PE fault tolerance; (c) mesh node fault tolerance; (d) group fault tolerance.

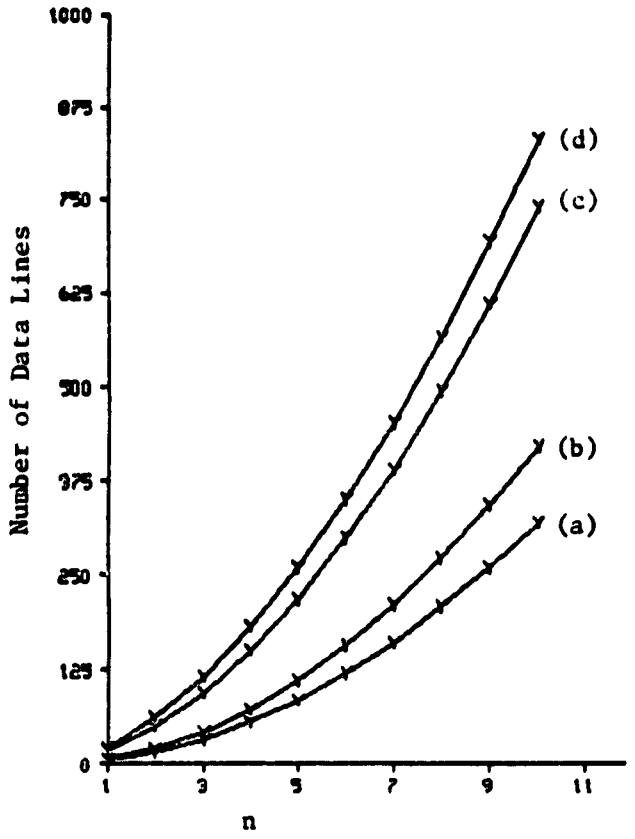


Fig. 14. Internal MIC data lines for an n x n active mesh; (a) no fault tolerance; (b) PE fault tolerance; (c) mesh node fault tolerance; (d) group fault tolerance.

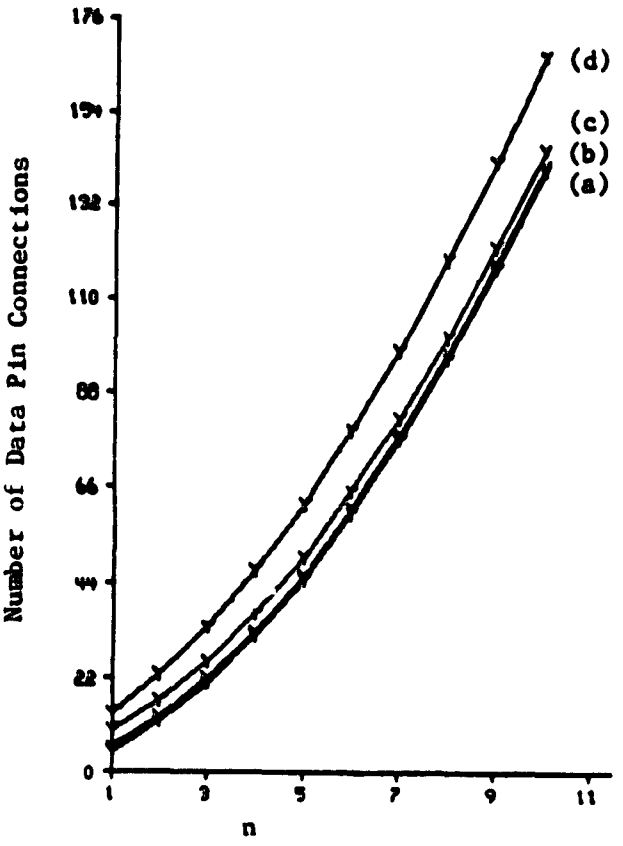
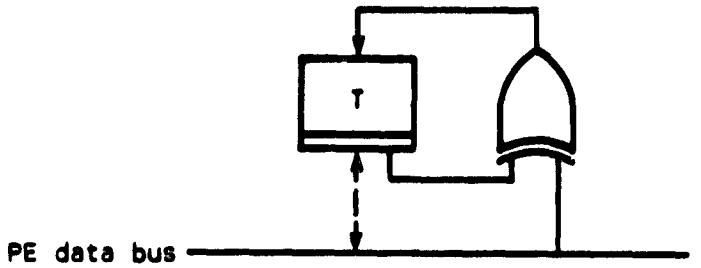
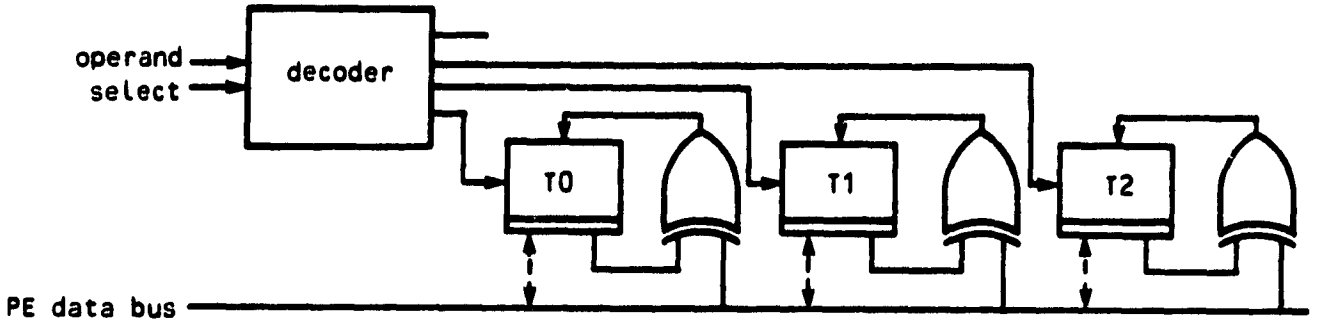


Fig. 15. MIC data pin connections for an n x n active mesh. (a) no fault tolerance; (b) PE fault tolerance; (c) mesh node fault tolerance; (d) group fault tolerance.



(a)



(b)

Fig. 16. Bit-serial parity hardware.
(a) for a PE with multi-bit registers.
(b) for a simple PE with 1-bit registers.

With simple bit-serial PE ALU's having only 1-bit registers it is necessary to interleave the fetching and storing of operand bits. For example, an integer add operation requires first the least significant bit of the operands to be read and the first bit of the result is stored, then the next least significant bits are dealt with and so on. To deal with this data flow the parity hardware shown in Fig. 16(b) may be used. Three parity registers are used in this case; T1 and T2 compute and check the parity of the two operands while T3 computes the parity bit for the result. Two extra pin connections are needed to each PE ALU chip to specify which operand the current bit on the data bus belongs to.

An important feature of tagging data with parity bits as described above is that, like other bit serial operations, the data format including the number of parity bits is completely user programmable. The cost of a large amount of parity checking is a reduction in effective local storage and an increase in processing time. The cost of the parity checking is proportional to the number of data bits associated with each parity bit. For 32-bit operands this cost is fairly small i.e. in the order of 3% loss of storage and increase in processing time while for 1-bit logical data this cost may be 100%. The user has the freedom to select where and how much parity checking is to be done.

Once an error has been detected, either through data parity or by running diagnostic programs for the PE's the host processor must reconfigure the PE array. It isolates the problem by finding which column, and when possible, the node PE which is the source of the error, and generates the bit masks, using the PE array when possible, to reconfigure the array.

6.10 Conclusion

The problem of fault tolerance in highly parallel mesh connected processors has been considered and methods of protecting against the most probable faults in the PE array have been proposed.

Fault tolerance at different levels has been considered. It has been shown that fault tolerance to the most error sensitive components, i.e. the functionally complex PE ALU's and local memory chips, may be achieved at a low cost at the local level. More extensive but less common errors such as catastrophic chip failures, broken command lines including a faulty OR bus line, usually need to be dealt with at the more expensive module or column level.

The cost of a high degree of fault tolerance can be achieved with a moderate amount of additional hardware. Such hardware may become a very important part of VLSI PE arrays having 1,000,000 or more nodes or in applications for smaller arrays in situations where high reliability and fault tolerance is necessary.

6.11 References

1. W. J. Bouknight, S.A. Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh and D. L. Slotnick, "The Illiac IV System," Proceedings of the IEEE, Vol. 60, No. 4, April 1972, pp. 369-388.

2. K. E. Batcher, "Design of a Massively Parallel Processor," IEEE Transactions on Computers, Vol. C-29, No. 9, September 1980, pp. 836-840.

3. P. A. Gilmore, K. E. Batcher, M. H. David, R. W. Lott and J. T. Burkley, "Massively Parallel Processor: Phase 1 Final Report," Goodyear Aerospace Technical Report, 1979.

4. A. P. Reeves, "Parallel Computer Architectures for Image Processing," 1981 International Conference on Parallel Processing, Bellaire, Michigan, August 25-28, 1981.

5. A. P. Reeves, "The Anatomy of VLSI Binary Array Processors," in Languages and Architectures for Image Processing, M.J.B. Duff and S. Levialdi eds., Academic Press 1981, pp. 267-274.

6. A. P. Reeves, "On Efficient Global Information Extraction Methods for Parallel Processors," Computer Graphics and Image Processing, Vol. 14, pp. 159-169, 1980.

7. CONCLUSIONS

Parallel Pascal, an initial high level language for the MPP, has been specified with surprisingly few extensions to the base Pascal Language. It has been implemented on conventional computers via a translator and, for the MPP, the front end of the compiler which generates Parallel P-Code has been developed. The suitability of the language notation has been demonstrated by program examples of typical MPP algorithms. Several useful algorithms have been developed for the MPP including fast median filtering and efficient, bit-level arbitrary function implementation.

Other high level languages have been specified for the MPP including a Parallel Fortran and a Parallel APL. The availability of the Parallel P-Code language and a code generator for Parallel Pascal should greatly simplify the construction of a compiler for these languages; only a front end which compiles the source language into Parallel P-Code is needed. If Parallel P-Code is used as common intermediate language then programs written in one language will be able to call functions and procedures written in a different language.

A considerable effort was made to carefully design the Parallel P-Code language. This intermediate language is at a higher level than conventional P-Code since it must deal with the more complex environment of a parallel matrix processor; i.e., a host processor and a PE array. Arrays and record data structures are described by descriptors rather than offsets so that the selection of the memory system on which they reside may be made by an optimizer or code generator. Code generators may be based on Parallel P-Code for many other parallel processors in addition to the MPP. The linear format of Parallel P-Code is a carry over from its P-Code compiler origins. The experience gained from developing Parallel P-Code suggests that for a future intermediate language a parse tree structure format might be more appropriate. This is because of the many different data aggregate structures which occur internally in a parallel language program.

Architectural extensions to the MPP PE design have been proposed which would greatly enhance the PE's performance. The construction of such PE's is feasible with todays VLSI technology; furthermore, much larger, fault tolerant PE arrays could now be constructed. The performance of the initial MPP for the benchmark image processing tasks will obviously be strictly limited by the completely inadequate input and output facility of the host VAX computer and disk. If the I/O problem is solved then the next bottleneck is the small 1 K bits of local PE memory. However, there is an important set of large scale scientific tasks which could be efficiently implemented on the MPP (with a larger PE local memory) which are so computation intensive that the current I/O speed would not be a problem. Without a larger local memory, a MPP with a high speed disk, large staging buffer and efficient spooling mechanism may offer an alternative solution for these tasks.

Pascal was chosen as the most suitable base language, however, it also has some problems. The major problems, which were inherited by Parallel Pascal, are (a) user defined functions and procedures are constrained by strong typing to operate only on a single specified array size and (b) there is no separate compilation facility.

The fixed array size problem is a very frustrating limitation that makes the implementation of general purpose library functions very difficult. There have been many solutions proposed for this problem, one of the best of which is the conformant array schema which has been proposed for the next Pascal standard. This feature allows the actual index ranges of an array passed as an argument to be determined at run-time. However, the rank (number of dimensions) of the arrays is still fixed at compile time. If this becomes a standard then it could be incorporated into Parallel Pascal without any problems. For efficient compilation a further minor restriction may be that the set of all possible subranges to be passed as arguments should be determinable at compile time. Other possibilities exist before a Pascal standard is

established; for example, a preprocessor could be used to make multiple variant copies of a procedure which is called with different size arguments.

The lack of a separate compilation facility means that libraries cannot be used in the usual way. Also, large programs take a long time to compile since all functions and procedures must be recompiled with every compilation. Several proposals have been made for an external function and procedure facility for Pascal but none has yet been accepted as a standard. Once a standard is developed it should not be difficult to incorporate into Parallel Pascal. Until a standard is developed a mechanism (preprocessor) should be developed for Parallel Pascal which enables the use of libraries and also permits library functions to deal with different sized arrays. It may be possible to store the libraries in Parallel P-Code form, ther much of the recompilation overhead for large programs can be avoided.

The Parallel Pascal implemented on the MPP will have the initial restriction that the last two dimensions of parallel arrays must have 128 elements to match with the MPP PE array size. The initial implementation of Parallel Pascal may have some further restrictions to simplify and speed the development of the MPP code generator. More work needs to be done in this area. An optimizer should be developed to make Parallel P-Code generated by the compiler more efficient for the MPP. Also other design options for the code generator should be explored such as the location of program code (MCU or VAX) and the prefetching of data from secondary storage. These options may be better explored once the initial code generator and the MPP are operational.

In order to make the MPP a more complete system which is convenient to use a library facility needs to be added to Parallel Pascal, as mentioned above, and a library of commonly used, efficiently coded functions needs to be developed. Initially this library could be very quickly established with

maximum flexibility by programming the functions in Parallel Pascal. Then key functions should be recoded in assembler language in order to achieve the maximum performance of the MPP. Since tne MPP will frequently be limited by the input/output requirements the library should include an I/O spooling system such as the one described in Section 2.

APPENDIX A: PARALLEL PASCAL SPECIFICATION

A.1 Overview

Parallel Pascal is a high-level language, based upon standard Pascal, for parallel matrix processors. The philosophy of the standard language was a major factor in the choice of extensions. In the following description of Parallel Pascal, familiarity with standard Pascal is assumed. [Standard Pascal is described in reference 1. A more recent definition is given in reference 2.]

A.2 Declarations

Each program, procedure, or function block in a Parallel Pascal program consists of a (possibly empty) set of declarations followed by a set of instructions. The declarations are grouped together according to their function: statement label definitions, constant definitions, type definitions, and variable definitions.

Parallel Pascal uses the same syntax as standard Pascal for

these definitions; however, two extensions are provided: subrange constants and parallel array types.

A.2.1 Constant Subranges

Standard Pascal uses the syntax

```
const
    identifier = value;
```

to associate ``value'' with the named ``identifier''. In standard Pascal, ``value'' must be either a literal or a (possibly signed) previously-defined constant identifier.

Parallel Pascal extends the definition of a constant to include a constant subrange. Constant subranges are used in array indexing (described below). Effectively the definition of an identifier as a constant subrange associates two values with the identifier - conceptually these represent a consecutive range of values. The syntax is:

```
const
    identifier = low .. high;
```

where ``low'' and ``high'' are either literals or (possibly signed) previously-defined constant identifiers. As an example:

```
const
    mpplow = 0;
    mpphigh = 127;
    mppidx = mpplow..mpphigh;
```

associates the integers 0 and 127 with the identifier ``mppidx''. When used in an array indexing expression, ``mppidx'' represents the ordered set of integers (0, 1, 2, ..., 126, 127).

A.2.2 Parallel Array Types

Standard Pascal specifies an array type definition with the syntax:

```
type
    newtype = array [ indextype ] of aeltype;
```

where ``newtype'' is the name of the new array type, ``indexrange'' is a type expression (either a subrange or a scalar type) defining the type of the indices, and ``aeltype'' is the type of the array elements.

On a parallel matrix processor, it is common to store some arrays on the non-parallel host machine (or in the scalar control unit) and some in the (parallel) hardware array. Parallel Pascal provides the reserved word parallel to allow the programmer to specify the memory in which an array should reside. A parallel array is defined with the syntax:

```
type
    newtype = parallel array [ indextype ] of aeltype;
```

Aside from the memory in which they reside, parallel arrays and ``ordinary'' arrays are treated identically in Parallel Pascal. The parallel keyword exists only to provide a means for the programmer to give the compiler a ``hint'' as to a variable's usage.

A.3 Array Expressions

The principle difference between standard Pascal and Parallel Pascal is that Parallel Pascal permits the specification of array expressions. In other words, arrays may be added, multiplied, compared, etc. as aggregate quantities rather than element-by-element. In order to deal with arrays, and sections of arrays, as aggregate units, Parallel Pascal provides extensions to standard Pascal's array indexing mechanisms.

As in standard Pascal, a scalar (non-array) expression may be used as an index. Optionally, a subrange constant may be added to the scalar expression. The subrange addition is specified by the special operator ``@'' to prevent ambiguity when a compiler (or human) is parsing the program. The subrange constant may either be an identifier defined with a const statement (see above) or a literal subrange - two constants separated by the symbol ``..''. If the scalar expression is zero it may be omitted. As an example, if the array ``m'' is defined by:

```
var
    m: array [1..10] of integer;
    i: integer;
```

then the expression

```
m[i@1..5]
```

specifies the following subset of ``m'':

```
m[i+1]  m[i+2]  m[i+3]  m[i+4]  m[i+5]
```

Finally, if it is desired to select the entire range of an index, the index expression may be omitted entirely. Hence, for ``m'' defined above, either of the expressions

m[] or m

will select the entire range of the array.

Parallel Pascal also provides a mechanism whereby the individual bits of an integer array element can be accessed. This mechanism is known as bit indexing. Since the form in which numbers are represented varies widely from machine to machine, bit indexing is inherently a very non-portable feature; however, the availablility of this feature may allow the programmer to avoid the use of assembly-language code which would be even less portable and more difficult to write, debug, and maintain. The bit index follows the ``regular'' indices and is preceeded by a colon:

```
arr[2,3:4] - select bit 4 of arr[2,3]
arr[:0] - select bit 0 of all elements of ``arr''
```

Bits are numbered from zero, with bit 0 considered the lowest-order bit.

In order to prevent ambiguity, when arrays are used together in an expression they must be conformable. (Additionally, the array elements must be type compatible, as in standard Pascal.) Two arrays are conformable if they have the same rank (number of dimensions) and the same shape. Additionally, if the index ranges of the arrays are not identical, then the non-matching

index range(s) of at least one of the arrays must be explictly specified. Table 1 illustrates the conformability of two arrays of the same size but with different index ranges.

Table 1: Conformability Examples

```
a: array [1..5] of integer;
b: array [0..4] of integer;
```

a	b	not conformable (implied ranges do not match)
a	b[@0..4]	conformable (explicit range for ''b'')
a[@1..2]	b	not conformable (shapes do not match)
a[@1..5]	b[@0..4]	conformable

A.4 Standard Functions and Procedures

A.4.1 Elemental Functions

Standard Pascal defines a number of standard functions to perform input/output, type conversion (e.g. truncating a real to an integer), and to perform common mathematical computations (e.g. cosine function). Parallel Pascal considers these functions to be ''generic'' in the sense that they may operate upon an array of any shape. For these functions, called ''elemental'' because they treat each element of the array independently, the value returned by the function is the same shape as the function argument. For example, given the definitions:

```
var
    sine: array [1..10] of real;
    angle: array [1..10] of real;
```

the following computes the sine function for each element of ``angle'' and stores the result in the corresponding element of ``sine'':

```
sine := sin(angle);
```

Table 2 summarizes the elemental functions.

Table 2: Elemental Functions

syntax	meaning
	type conversions
trunc(x)	truncate real to integer
round(x)	round real to integer
ord(x)	ordinal value of x (for scalar types)
chr(x)	character with ordinal value x
	arithmetic functions
abs(x)	absolute value
sqr(x)	square (i.e. x^2)
sqrt(x)	square root (i.e. $\sqrt{x}$)
exp(x)	exponential (i.e. e^x)
ln(x)	natural logarithm
sin(x)	sine function
cos(x)	cosine function
arctan(x)	arctangent function
	miscellaneous
odd(x)	boolean: true if x is odd
eof(f)	boolean: true if at end-of-file on file f
eoln(f)	boolean: true if at end-of-line on file f
succ(x)	successor of x (if defined)
pred(x)	predecessor of x (if defined)

A.4.2 Transformational Functions

In addition to the elemental functions, Parallel Pascal also provides some ``transformational'' functions, so named because they perform transformations upon the entire array rather than element-by-element. Table 3 summarizes the transformational functions, which are discussed in more detail below.

Table 3: Transformational Functions

syntax	meaning
shift(array, S1, S2, ..., Sn)	end-off shift data within array
rotate(array, S1, S2, ..., Sn)	circularly rotate data within array
expand(array, dim, size)	expand array along specified dimension
transpose(array, D1, D2)	transpose two dimensions of array
sum(array, D1, D2, ..., Dn)	reduce array with arithmetic sum
prod(array, D1, D2, ..., Dn)	reduce array with arithmetic product
all(array, D1, D2, ..., Dn)	reduce array with Boolean AND
any(array, D1, D2, ..., Dn)	reduce array with Boolean OR
max(array, D1, D2, ..., Dn)	reduce array with arithmetic maximum
min(array, D1, D2, ..., Dn)	reduce array with arithmetic minimum

The functions ``shift'' and ``rotate'' are used to move data within an array. These two functions have the same syntax; they differ in that ``shift'' performs an end-off shift of the array (with zeros shifted in at the other end) whereas ``rotate'' performs a circular rotation along the specified dimensions. The function call specifies the array to be operated upon and the amount that each dimension is to be moved. As an example, given the definition

```
var
    a,b: array [0..127, 0..127] of integer;
```

the statement

```
a := shift(b, 0, 3);
```

is functionally equivalent to (but much faster than):

```
for i := 0 to 127 do
  begin
    for j := 0 to 124 do
        a[i,j] := b[i,j+3];
    for j := 125 to 127 do
        a[i,j] := 0;
  end;
```

The ``transpose'' function is used to transpose an array about two specified dimensions. If only one dimension is specified, the array is ``flipped'' about that dimension. In order to determine the shape of the result at compile-time, the dimensions about which the transposition are to take place must be specified by compile-time constants.

The shape of an array may be altered by the ``expand'' function. The arguments to this function are the array to be operated upon, the new dimension along which the expansion is to take place, and a type specification. The array is expanded along the indicated dimension. If the rank of the array is n, then the second argument to ``expand'' can be at most n+1. The dimension along which the expansion is to take place must be a compile-time constant, in order to ensure that the shape of the result can be determined at compile-time.

The functions ``sum'', ``prod'', ``all'', ``any'', ``max'', and ``min'' are used to reduce an array along a specified set of

dimensions. These functions differ only in the reduction operation that they perform. The arguments to a reduction function are the array to be operated upon an a list of dimensions over which the reduction is to be performed. In order to ensure that the compiler can determine the shape of the result, the dimensions must be compile-time constants. The first dimension of an array is numbered 1 (not 0). As an example, given a two-dimensinal array ``a'',

```
sum(a,1,2)
```

computes the arithmetic sum of all of the elements in ``a'', while

```
max(a,2)
```

produces a vector consisting of the maximum element in each row of ``a''.

A.4.3 Standard Procedures

Like standard Pascal, Parallel Pascal also provides a set of standard procedures for file handling, dynamic memory allocation, and data transfer. Table 4 summarizes the available standard procedures.

Table 4: Standard Procedures

syntax	meaning
	file handling procedures
put(f)	append the buffer variable to file f
get(f)	get a new buffer variable from file f
reset(f)	reset file f for reading (rewind)
rewrite(f)	prepare file f for writing
	dynamic memory allocation
new(p)	allocate storage, place address in p
new(p,t1,...tn)	as above, but fix record variants
dispose(p)	release storage described by p
	data transfer procedures
pack(a,i,z)	pack i elements of a into z
unpack(z,a,i)	unpack i elements of z into a

A.5 Control Flow

In addition to the standard Pascal control structures (if, case, while, repeat-until, goto), Parallel Pascal provides the where statement for conditional assignment to arrays according to a controlling expression. The syntax is

```
where array-expression do
    statement
otherwise
    statement
```

where the otherwise and the second controlled statement may be omitted.

The execution of a where is defined as follows. First, the controlling expression is evaluated to obtain a Boolean array (mask array). Next, the first controlled statement is evaluated. Array assignments are masked according to the mask array computed

above. Finally, if thereis a second controlled statement, it is evaluated. Array assignments within the second controlled statement are masked by the inverse of the mask array.

where statements may be nested, provided that all of the controlling array expressions are conformable and type compatible. The effect of a where statement is local to the procedure or function in which it appears - it does not affect the execution of any procedures of functions called from one of the controlled statements.

A.6 Parallel Pascal Grammar

The metalanguage for this grammar is as follows:

1. The left-hand-side of each production is separated from its right-hand-side by the symbol ::= .

2. Nonterminal names are represented directly.

3. Literal symbols are underlined. In cases where confusion with metasymbols is possible, literals are enclosed in double-quote marks ".

4. The vertical bar | represents a choice between alternatives.

5. Parentheses () enclose a selection of constructions which are separated by vertical lines.

6. Square brackets [] enclose a construction or choice of constructions which may occur zero or one times in the production.

7. Curved brackets {} enclose a construction or choice of constructions which may occur any number of times.

letter ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

digit ::= 0|1|2|3|4|5|6|7|8|9

special-symbol ::= and|array|begin|case|const|div|do|downto|else|end|
file|for|function|goto|if|in|label|mod|nil|not|
of|packed|parallel|procedure|program|record|repeat|
set|then|to|type|until|var|while|with

identifier ::= letter { letter | digit }

```
directive ::= letter { letter | digit }

digit-sequence ::= digit { digit }
unsigned-integer ::= digit-sequence
unsigned-real ::=
     unsigned-integer . digit-sequence [ e scale-factor ] |
     unsigned-integer e scale-factor
unsigned-number ::= unsigned-integer | unsigned-real
scale-factor ::= signed-integer
sign ::= +|-
signed-integer ::= [ sign ] unsigned-integer
signed-real ::= [ sign ] unsigned-real
signed-number ::= signed-integer | signed-real

label ::= digit-sequence

character-string ::= ' string-element { string-element } '
string-element ::= apostrophe-image | string-character
apostrophe-image ::= ''
string-character ::= one-of-an-implementation-defined-set-of-characters

block ::= label-declaration-part
     constant-definition-part
     type-definition-part
     variable-declaration-part
     procedure-and-function-declaration-part
     statement-part

label-declaration-part ::= [ label label { , label } ; ]

constant-definition-part ::= [ const constant-definition ;
     { constant-definition ; } ]

type-definition-part ::= [ type type-definition ;
     { type-definition ; }

variable-declaration-part ::= [ var variable-declaration ;
     { variable-declaration ; } ]

procedure-and-function-declaration-part ::=
     { ( procedure-declaration | function-declaration ) ; }

statement-part ::= compound-statement

constant-definition ::= identifier = constant
constant ::= subrange-constant | scalar-constant
subrange-constant ::= scalar-constant .. scalar-constant |
     subrange-constant-identifier
scalar-constant ::= [ sign ] ( unsigned-number | scalar-constant-identifie
     character-string
scalar-constant-identifier ::= constant-identifier
subrange-constant-identifier ::= constant-identifier
constant-identifier ::= identifier
```

```
type-definition ::= identifier = type-denoter
type-denoter ::= type-identifier | new-type
new-type ::= simple-type | structured-type | pointer-type

simple-type-identifier ::= type-identifier
structured-type-identifier ::= type-identifier
pointer-type-identifier ::= type-identifier
type-identifier ::= identifier

simple-type ::= ordinal-type | real-type
ordinal-type ::= enumerated-type | subrange-type | integer-type |
     Boolean-type | char-type | ordinal-type-identifier

enumerated-type ::= "(" identifier-list ")"
identifier-list ::= identifier { , identifier }

subrange-type ::= scalar-constant .. scalar-constant

structured-type ::= [ packed ] unpacked-structured-type |
     structured-type-identifier

unpacked-structured-type ::= array-type | record-type |
     set-type | file-type

array-type ::= [ parallel ] array "[" index-type { , index-type } "]"
     of component-type
index-type ::= ordinal-type
component-type ::= type-denoter

record-type ::= record [ field-list [ ; ] ] end
field-list ::= fixed-part [ ; variant-part ] | variant-part
fixed-part ::= record-section { ; record-section }
variant-part ::= case variant-selector of variant [ ; variant ]
variant-selector ::= [ tag-field : ] tag-type
tag-field ::= identifier
variant ::= case-constant-list : "(" [ field-list [ ; ] ] ")"
tag-type ::= ordinal-type-identifier
case-constant-list ::= case-constant { , case-constant }
case-constant ::= constant

set-type ::= set of base-type
base-type ::= ordinal-type

file-type ::= file of component-type

pointer-type ::= ↑ domain-type | pointer-type-identifier
domain-type ::= type-identifier

variable-declaration ::= identifier-list : type-denoter

variable-access ::= entire-variable | component-variable |
     referenced-variable | buffer-variable
```

```
entire-variable ::= variable-identifier
variable-identifier ::= identifier

component-variable ::= indexed-variable | field-designator

indexed-variable ::= array-variable "["
     [ index-expression { , index-expression } ] [ : bit-specifier ] "]"
index-expression ::= [ expression ] [ @ subrange-constant ]
array-variable ::= variable-access
bit-specifier ::= simple-expression

field-designator ::= record-variable . field-designator
record-variable ::= variable-access
field-identifier ::= identifier

buffer-variable ::= file-variable ↑
file-variable ::= variable-access

referenced-variable ::= pointer-variable ↑
pointer-variable ::= variable-access

procedure-declaration ::=
     procedure-heading ; directive |
     procedure-identification ; procedure-block |
     procedure-heading ; procedure-block
procedure-heading ::= procedure identifier [ formal-parameter-list ]
procedure-identification ::= procedure procedure-identifier
procedure-identifier ::= identifier
procedure-block ::= block

function-declaration ::=
     function-heading ; directive |
     function-identification ; function-block |
     function-heading ; function-block
function-heading ::=
     function identifier [ [ formal-parameter-list ] : result-type ]
function-identification ::= function function-identifier
function-identifier ::= identifier
result-type ::= type-identifier
function-block ::= block

formal-parameter-list ::=
     "(" formal-parameter-section { ; formal-parameter-section } ")"
formal-parameter-section ::=
     value-parameter-specification |
     variable-parameter-specification |
     procedural-parameter-specification |
     functional-parameter-specification
value-parameter-specification ::= identifier-list : type-identifier
variable-parameter-section ::= var identifier-list : type-identifier
bound-identifier ::= identifier
procedural-parameter-specification ::= procedure-heading
functional-parameter-specification ::= function-heading
```

```
unsigned-constant ::= unsigned-number | character-string |
     constant-identifier | nil

factor ::= variable-access | unsigned-constant | bound-identifier |
     function-designator | set-constructor |
     "(" expression ")" | not factor
set-constructor ::= "[" [ member-designator { , member-designator }] "]
member-designator ::= expression [ .. expression ]
term ::= factor { multiplying-operator factor }
simple-expression ::= [ sign ] term { adding-operator term }
expression ::=
     simple-expression [ relational-operator simple-expression ]

multiplying-operator ::= * | / | div | mod | and
adding-operator ::= + | - | or
relational-operator ::= = | <> | < | > | <= | >= | in

function-designator ::= function-identifier [ actual-parameter-list ]

actual-parameter-list ::= "(" actual-parameter { , actual-parameter } " "
actual-parameter ::= expression | variable-access |
     procedure-identifier | function-identifier

statement ::= [ label : ] ( simple-statement | structured-statement )

simple-statement ::= empty-statement | assignment-statement |
     procedure-statement | goto-statement
empty-statement ::=

assignment-statement ::=
     ( variable-access | function-identifier ) := expression

procedure-statement ::= procedure-identifier [ actual-parameter-list ]

goto-statement ::= goto label

structured-statement ::= compound-statement | conditional-statement |
     repetitive-statement | with-statement

compound-statement ::= begin statement-sequence end
statement-sequence ::= statement { ; statement }

conditional-statement ::= if-statement | case-statement |
     where-statement

if-statement ::= if Boolean-expression then statement [ else-part ]
else-part ::= else statement

case-statement ::= case case-index of case-list-element
     { ; case-list-element } [ ; ] end
case-list-element ::= case-constant-list : statement
case-index ::= expression
```

```
where-statement ::= where parallel-Boolean-expression do
     statement [ otherwise-part ]
otherwise-part ::= otherwise statement
parallel-Boolean-expression ::= expression

repetitive-statement ::= repeat-statement | while-statement |
     for-statement

repeat-statement ::= repeat statement-sequence until Boolean-expression

while-statement ::= while Boolean-expression do statement

for-statement ::= for control-variable := initial-value
     ( to | downto ) final-value do statement
control-varible ::= entire-variable
initial-value ::= expression
final-value ::= expression

with-statement ::= with record-variable-list do statement
record-variable-list ::= record-variable { , record-variable }

read-parameter-list ::= "(" [ file-variable , ] variable-access
     { , variable-access } ")" ]

readln-parameter-list ::= [ "(" ( file-variable | variable-access )
     { , variable-access } ")" ]

write-parameter-list ::= "(" [ file-variable , ] variable-parameter
     { , write-parameter } ")"
write-parameter ::= expression [ : expression [ : expression ] ]

writeln-parameter-list ::= [ "(" ( file-variable | write-parameter )
     { , write-parameter } ")" ]

program ::= program-heading ; program-block .
program-heading ::= program identifier [ "(" program-parameters ")" ]
program-parameters ::= identifier-list
program-block ::= block
```

A.7 Parallel Pascal Error Codes

In addition to the standard Pascal error codes (defined in reference 1), the following error codes are defined for Parallel Pascal:

350: must be parallel array type
351: illegal type for parallel array
352: boolean type required
360: arrays not compatible
361: array not compatible with controlling array
362: result must be array type
363: parallel array not allowed
364: function result type must be array
365: dimension not compatible with array
366: integer constant expected
367: at least one dimension expected
368: bit index type must be integer
369: error in number of standard function arguments
370: subrange exceeds array index limits
371: set type not compatible with array index type
373: bit indexing not allowed
374: illegal array type for bit indexing
375: subrange constant expected
397: unimplemented feature
398: implementation restriction
399: implementation restriction
400: internal inconsistency

A.8 References

1 Kathleen Jensen and Niklaus Wirth, PASCAL User Manual and Report, Springer-Verlag, Berlin, Heidelberg, New York (1974).

2 A. M. Addyman, "A Draft Proposal for Pascal," ACM SIGPLAN Notices Vol. 15(4), pp.1-66 (April 1980).

APPENDIX B: PARALLEL P-CODE SPECIFICATION

B.1 Data Declarations

To permit efficient handling of arrays, both parallel and ordinary, it is necessary that the code generator be supplied with information about the size and shape of data items. Thus, the intermediate language must include specifications for the fundamental data types (arrays and records).

The code generator's view of the world is based upon the following assumptions:

1. The code generator ``knows'' whether the code it is generating is to reside on the host (for the MPP this would be the VAX) or the sequential control unit.

2. A few standard types are predefined:

```
integer
real
Boolean
char
scalar pointer
array (i.e. parallel) pointer
file
```

All other types that the code generator must deal with are

defined in various ``pseudo-op'' statements in the intermediate language.

3. The size and shape of arrays and the layout of records are known to the code generator.

The compiler specifies the target machine on which code is to be generated via the .SITE pseudo-operator. The syntax is:

```
.SITE     sitename
```

where sitename may be either ``HOST'' to specify the host processor (for the MPP this is the VAX-11/780) or ``MCU'' to specify the (main) control unit of the parallel processor.

The following pseudo-operators are used to define derived types:

.ARRAY This pseudo-operator is used to specify the size and shape of an array type. The syntax is:

```
.ARRAY newname,basetype,rank,dim0low,dim0high,...
```

where newname is the name of the type which is being defined, basetype is the name of a previously defined type, rank is the total number of dimensions of the array, and dimilow dimihigh are the low and high bounds of each index range. rank is negative if the array is a parallel array. For instance, the type definitions:

```
type
     parr = parallel array [1..128,1..128] of integer;
     arr  = array [5..10] of parr;
```

would be translated to

```
.ARRAY parr,integer,-2,1,128,1,128
.ARRAY arr,parr,1,5,10
```

.RANGE This pseudo-operator is used to define a subrange. The syntax is:

.RANGE newname,low,high

For example, the definition

```
type
    xxx = 10..32;
```

would be translated to

.RANGE xxx,10,32

.RECORD This pseudo-operator is used to define a record. The code generator must know the configuration of a record, hence it is necessary to provide the type of each component. The code generator is responsible for computing the appropriate offset. The syntax is:

.RECORD recname,cmpname,offset,type

where recname is the name of the record type, cmpname is the name of the current component, offset is the offset (see below) and type is the type of component. There is one recname field per record and one cmpname for each component. offset is normally ''nil'', indicating that the code generator should choose the offset (normally as the next component in the record

being defined). If offset is not ``nil'' it specifies a record component; in this case, the current record component is to have the same offset as the named component. This is used to align variant records. A .RECORD statement implicitly defines two types: the record itself and the record component. As an example, the type definition:

```
type
    rec = record
            x: integer;
            y: real;
            case Boolean of
              false:zf: integer;
              true: zt: real;
            end;
```

could be translated as follows

```
.RECORD   rec,x,nil,integer
.RECORD   rec,y,nil,real
.RECORD   rec,zf,nil,integer
.RECORD   rec,zt,zf,real
```

(The last definition specifies that the component ``zt'' is to aligned with the component ``zf''.)

.SET This pseudo-operator is used to specify the size of a power set. The syntax is:

```
.SET newname,low,high
```

where newname is the name of the type which is being defined, low is the lowest element (integer) and high is the highest element. Sets of type char are converted by the compiler to the appropriate integer type.

.FILE This pseudo-operator is used to specify a file. The syntax is:

.FILE newname,ftype

newname is defined to be a file of ftype.

.POINT This pseudo-operator is used to defined pointer types. For the most part, pointers are considered to be the same thing as integers; however, occasionally it is necessary to distinguish them. The syntax is:

.POINT newname,ptype

which causes newname to represent a pointer to type ptype.

.TYPE This pseudo-operator equivalences an existing type-name with a new name. The syntax is:

.TYPE newname,oldname

newname is defined to be the same type as oldname. This redundant statement allows some simplification in the front-end of the Parallel Pascal compiler.

The intermediate language representation of arrays consists of two portions. The first is the (static) logical type information specified by the .ARRAY pseudo-operator. The second is the (dynamic) information about the physical storage allocation which is required at runtime. If set and vector indexing are excluded, then Parallel Pascal permits any contiguous subset of array elements to be operated upon at once.

The intermediate language manipulates arrays through a conceptual entity called an array descriptor. Instead of pushing the actual elements of an array onto the stack, the descriptor for that array is pushed instead. The array descriptor specifies the base address and the storage mapping defined by the index ranges of each array dimension. The compiler does not know or care what format the array descriptor has. The LLA instruction is used to load a ``blank'' descriptor onto the stack (a descriptor specifying the address but no index ranges); a sequence of indexing instructions (IX0, IX1, IX2) is performed to ``fill in'' this information.

Records are similarly defined by a record descriptor. Like array descriptors, record descriptors consist of the (static) information provided by the .RECORD pseudo-operator and the (dynamic) information contained on the runtime stack. The dynamic information specifies the address of the record and the fields of the record which have been selected to participate in a future operation (e.g. load, add, store). The compiler does not know or care about the format of this information. A record descriptor is constructed by performing an LLA (which loads a descriptor for the entire record) followed by one or more SEL instructions to select successively-nested fields.

Records may contain arrays and array elements may be records. The appropriate combination of IX? and SEL instructions is used (recursively) to select a set of array elements within a record and a field within a set of nested

records.

Most of the intermediate language operators require the specification of a type. The following code segment illustrates how the (static) type and (dynamic) array descriptor are used:

```
type
    arr1 = array [1..5] of integer;
    arr2 = array [2..6] of integer;
var
    a: arr1;
    b: arr2;
begin
    a := b[1@1..5];
end.
```

```
.ARRAY    arr1,integer,1,1,5
.ARRAY    arr2,integer,1,2,6
IX0
LLA  <address of ``a''>
IX0  arr1
LLA  <address of ``b''>
LDC  integer 2
LDC  integer,6
IX2  arr2
LDI  arr2
STO  arr1
```

An example with records and record descriptors (the ``xxx'' and ``yyy'' are arbitrarily-chosen names):

```
type
    arr10 = array [1..10] of real;
    recrd = record
                x, y: arr10;
            end;
    arrrec = array [1..3] of recrd;

var
    v: arrrec;

    v.x[5] := 0;
```

```
.ARRAY     arr10,real,1,1,10
.RECORD    recrd,x,nil,arr10
.RECORD    recrd,y,nil,arr10
.ARRAY     arrrec,recrd,1,1,5

LLA  <address of ''v''>
IX0  arrrec
.ARRAY     xxx,real,1,1,5
SEL  arrrec,x,xxx
LDC  integer,5
.POINT     yyy,real
IX1  xxx,yyy
LDC  integer,0
CVT  integer,real
STO  real
```

B.2 Procedure/Function Arguments and Local Variables

The following pseudo-operators are used to define subroutine arguments and local variables:

.ENTRY This pseudo-operator indicates that a new block is being entered.

.EXIT This pseudo-operator complements .ENTRY by indicating the end of a block. Definitions in the current block are to be ''forgotten'' by the code generator at this point.

.ARG This pseudo-operator defines an argument to the current subroutine. The syntax is:

.ARG num,type,rv

where num is an integer which starts at zero (see comment below) and is incremented by one for each

argument, type is the type of the argument, and rv is either zero or non-zero to indicate that the data is being passed by value or reference, respectively. The num field normally is a positive integer. If the subroutine returns a value (i.e. if it is a function rather than a procedure) the space for the result is reserved by an .ARG pseudo-operator with zero in the num field.

.LOCAL This pseudo-operator defines local variables. The syntax is:

.LOCAL num,type,equ

where num and type are defined as for .ARG. equ is used to indicate storage sharing to the code generator. If equ is zero, the next available memory location should be allocated. If equ is non-zero it specifies a previously-defined local variable; in this case the storage for the new local variable is to be allocated on top of the previously-defined variable. The numbers assigned to local variables belong to the same space as those assigned to subroutine arguments. Thus, if there are n arguments to a subroutine (and hence n .ARG statements), the num field for the first .LOCAL statement will contain n+1.

C.3 Parallel Pcode Mnemonics

The mnemonics and their functions for the opcodes defined in parallel p-code are as follows:

ABS Produce absolute value. The syntax is:

ABS type

ADD Add two operands. The syntax is:

ADD type

AND Perform Boolean ``and''. This is only defined for Boolean variables; however, an array may be specified so a type is required. The syntax is:

AND type

CHK Check that top of stack is between two specified values.

CSP Call standard procedure. The syntax is

CSP procedurename,argtype,resulttype

where ``argtype'' is the type of the primary argument and ``resulttype'' is the type of the function result. (If the called routine is a standard procedure the literal string ``nil'' is used.) Calls of standard procedures and functions are discussed in more detail below.

CUP Call user procedure. The syntax is

CUP level,procedurename,resulttype

Calls of user procedures and functions are discussed in more detail below.

CVT Convert the top of stack from one type to another. Conversions performed by this operator may alter both the shape (array dimensions) and the underlying type of the object. The syntax is:

CVT oldtype, newtype

CVN Convert the next-to-top of stack from one type to another. This is similar to CVT, defined above. The syntax is:

CVN oldtype, newtype, tstype

where tstype is the type of item on top of the stack (this information is required in order to locate the next-to-top element, since descriptors vary in size).

DEC Decrement top of stack by a specified amount. This may only be applied to integers or subranges or arrays of type integer or subrange. The syntax is

DEC type,amount

DIF Evaluate set difference. The syntax is:

DIF type

DIV Perform real division. The syntax is:

DIV type

Notice that, unlike Pascal P4, integer operands must be explicitly converted to real format before the division.

DUP Duplicate top of expression stack. The syntax is

DUP type

DVI Perform integer division. This may only be applied to integers or subranges or arrays of type integer or subrange. The syntax is:

DVI type

ENW This operator is used to remove the effect of a mask (``end where''). The syntax is:

ENW type

where type is the type of the mask (located on top of the expression stack). The mask stack is ``popped''; the previous mask (that is, the mask in effect before the most recent WHR) is restored. This operation is illegal if there is no current mask.

EOF Test for end-of-file condition. There are no arguments, the filename is assumed to be the top item on the stack.

EQU Test for equality. The syntax is:

EQU type

FJP Jump if item on top of stack is false. The item must be a scalar Boolean quantity. The syntax is

FJP label

GEQ Test for greater-than or equal-to. The syntax is:

GEQ type

GRT Test for greater-than. The syntax is:

GRT type

INC Increment top of stack by specified amount. This may only be applied to integers or subranges or arrays of type integer or subrange. The syntax is

INC type,amount

INN Test for set membership. The syntax is

INN settype

INT Perform set intersection. The syntax is

INT settype

IOR Perform Boolean inclusive or. This is only defined for Boolean variables; however, an array may be specified so a type is required. The syntax is

IOR type

IXO Index with zero values. This operator is used in the construction of array descriptors. It ``fills in'' the index specification for the first unspecified index range

in the array descriptor on the top of the runtime stack. The syntax is

IX0 type

IX1 Index with one value. This operator is used in the construction of array descriptors. The top of stack is an array descriptor for which at least one dimension is unspecified. This instruction selects one value for the first unspecified dimension. This reduces the dimension of the array by one The descriptor on top of the stack will be modified to reflect this new type; if the logical type is now a scalar this descriptor will (conceptually) be a pointer to a scalar. The syntax is

IX1 oldtype,newtype

where ``oldtype'' is the type of the descriptor on top of the stack before indexing and ``newtype'' is the type after indexing.

IX2 Index with two values. This operator is used in the construction of array descriptors. It ``fills in'' the index specification for the first unspecified index range. The second element on the stack and the top of stack are integers specifying the low and high bounds, respectively. The third element on the stack is the array descriptor. The syntax is:

IX2 type

LCA Load address of constant. The syntax is:

LCA type,constant

The constant itself is specified, the code generator is responsible for setting up a static constant somewhere. The constant must be a scalar.

LDC Load constant. The syntax is

LDC type, constant

LDI Load indirect (load value pointed to by top of stack). The syntax is

LDI type

When an LDI is applied to a file, the file buffer is loaded onto the stack.

LEQ Test for less-than or equal-to. The syntax is

LEQ type

LES Test for less-than. The syntax is

LES type

LLA Load address. The syntax is:

LLA lexlevel, localid

where lexlevel is the lexical level (the level of nesting) and localid is the local variable index number as specified by a .LOCAL or .ARG definition (see above).

LOD Load contents of address. The syntax is

LOD type,lexlevel,localid

where type is the type of the variable, lexlevel is the lexical level (level of nesting), and localid is the local variable index as defined by a .LOCAL or .ARG definition (see above).

MOD Perform modulus (remainder) operation. This is only valid for items of type integer (or subrange) or arrays of type integer (or subrange). The syntax is:

MOD type

MOV Move a specified number of storage units. The syntax is still unknown.

MUL Multiply. The syntax is:

MUL type

MST Mark stack (used for procedure calls). The syntax is

MST level

where level is the lexical level of the procedure or function which will be called.

NEG Negate top of stack. The syntax is

NEG type

NEQ Test for not-equal. The syntax is

NEQ type

NOT Perform Boolean not (logical complement). This is only valid for Boolean scalars and arrays. The syntax is

NOT type

ODD Test for odd. This is only valid for integer (or subrange) scalars and arrays. The syntax is

ODD type

OTW This operator implements the ''otherwise'' conditional. It is used to reverse the sense of a nested mask established by the WHR instruction. The syntax is:

OTW type

where type is the type of the current mask. (At this point, the expression on top of the expression stack specifies the current mask.) OTW is illegal if no mask is currently in effect. If a mask is currently in effect, the new mask is computed by performing an exclusive-or between the current mask and the previous mask (that is, the mask that was in effect before the most recent WHR).

RET Return from block. The syntax is:

RET type

where type is either the literal string ''nil'' or is a type name. In the former case, the called routine is a procedure and no value is to be returned to the caller. In

the case of a function, type is the type of the data which is returned by the function. (See below.)

SEL Select a record field. This operator causes a record descriptor to be constructed from an address, array descriptor, or record descriptor already on the stack. The syntax is:

SEL oldtype,cmptype,newtype

where oldtype is the type of the current top-of-stack, cmptype is the type of the item being selected (i.e. it specifies the record type and the component name), and newtype is the type of the result.

SUB Perform subtraction. The syntax is

SUB type

SGS Generate singleton set. The syntax is:

SGS settype

The set is constructed from the element on top of the stack.

SQA Square top of stack. The syntax is

SQA type

STO Store indirect (at address specified by second element on stack). The syntax is:

STO type

When a STO is applied to a file, the top of stack is stored in the file buffer.

STP Stop execution. There are no arguments.

STR Store at compile-time known address. The syntax is

STR type,lexlevel,localid

where type is the type of the value, lexlevel is the lexical level (level of nesting), and localid is the local variable index as defined by an .ARG or .LOCAL definition (see above).

UJC Error in case statement (abort). The syntax is

UJC label

UJP Unconditional jump. The syntax is

UJP label

UNI Perform set union. The syntax is:

UNI type

WHR Define a new logical mask (``where''). The syntax is:

WHR type

where type must be an array of type Boolean. Masking is performed in a nested manner. If there is no active mask, the top of the expression stack defines the mask. If a mask is active, the current mask is logically ANDed with

the array on the top of the expression stack to form the new mask. (The previous mask is ''pushed''.) In this case, the two expressions must have identical types. Masks established in this fashion can be further manipulated with the OTW and ENW operators.

XJP Indexed jump (jump to specified value + top of stack). The item on the top of the stack must be an integer or subrange. The syntax is

XJP value

B.4 Descriptors and the Stack

As in P4, all operations are performed on a conceptual stack. However, the stack may contain several different types of items beyond those allowed by P4. The representation of the various data types is described below:

scalars Scalars are manipulated directly on the stack, i.e. when a scalar is loaded the value of the scalar is placed on the stack. Thus, scalars can be directly manipulated.

arrays Unlike scalars, arrays are not pushed on the stack. Instead, the array is described by an array descriptor. The array descriptor consists of the type of the array (as defined by a .ARRAY statement) which is static, and

a dynamic portion which specifies the array address and the indexing information. The descriptor is constructed by performing an LLA (loading the ''lexical address'' of a variable) or CVT (when a scalar is converted to an array) followed by a series of indexing instructions (IX0, IX1, IX2) to specify the index ranges.

records — Records are represented by a record descriptor whose exact format is the domain of the code generator. The layout of the various records is specified by .RECORD pseudo-ops and the descriptors themselves are constructed by use of the SEL instruction.

sets — Sets are represented by a set descriptor which merely gives the address of the set. The permissible values in the set are known statically and are given by .SET statements.

Unlike the use of scalars, performing a ''load'' of an array, record, or set does not place the data on the stack. Instead, the hypothetical stack machine (machine code generator) replaces the descriptor on top of the stack with another which is identical except for the address (the address of a temporary array location is used). Similarly, when the conversion operator (CVT or CVN) is applied to convert one item to another, the conversion is performed into a temporary area and the appropriate

descriptor is placed on the stack.

It is illegal to perform any operation in which the size and shape of the arguments do not match. It is possible, however, that two arrays with different index ranges (but identical shapes) will be combined by an operation. In this case, the result will be placed into a temporary array according to the indices specified by the second descriptor on the stack. Also, an array consisting of a record component may be combined with another array, provided that the base types and array shapes match.

The function of the CVT and CVN operators when one of the types is a scalar deserves some comment. First, either may be used to convert a scalar to an array with the same type as the scalar. For instance,

```
.ARRAY    arr,real,1,1,5 (array [1..5] of real)
CVT  real,arr
```

In this case, the scalar on top of the stack is replaced by an array descriptor (which references a temporary area). This descriptor is ``blank'' - no indexing information is specified. A sequence of IX? instructions is then performed to ``fill in'' the indexing information. Second, either may be used to convert a one-element subset of an array into a scalar. In this case, the array descriptor on the stack is replaced by the appropriate scalar value.

B.5 Function and Procedure Calls

Standard procedures and functions (hereafter called subroutines) are all called in the same fashion. First the stack is marked with MST, specifying that the called routine is at lexical level zero. (The lowest lexical level available to user routines is 1, and that is used for the main program.) Next, the arguments are evaluated left-to-right and placed on the stack. Finally, the routine is called with a CSP instruction. In all standard function and procedure calls at most one argument has a type which varies from call to call; this is referred to as the ``primary'' argument. In addition to specifying the called routine, the CSP instruction specifies the type of the primary argument and the result type of the function (``nil'' if the called routine is a standard procedure).

Scalars are passed to subroutines on the stack; structured types are passed via descriptors. For call by reference, the array, set, or record descriptor is pushed on the stack. For call by value a LDI is performed - this causes the array to be copied into a temporary area and a descriptor for this temporary array to be placed on the stack.

Control is returned to the caller when the RET instruction is executed. If the called routine is a procedure, the runtime stack is reset to the last marked location. If the called routine is a function, the returned value is placed on the runtime stack in the locations reserved for it (see above) and

the stack is reset to the end of this area.

User subroutines are called in the same fashion as the standard ones except that the CUP instruction is used. This instruction specifies the lexical level at which the user subroutine will run and (for functions) the data type of the function result. (This is ``nil'' for procedures.)

B.5.1 Elemental Functions

The Pascal standard functions all may operate on either scalars (as in standard Pascal) or arrays. The result has the same shape as the function argument (although sometimes the base type is different). Because they operate independently upon the elements of arrays these functions are referred to as ``elemental functions''. The following functions form this set: abs, arctan, chr, cos, eof, eoln, exp, ln, odd, ord, pred, round, sin, sqr, sqrt, and trunc. For all of these functions, the stack is marked, the argument is loaded, and the function is called:

```
MST  0
<load argument>
CSP  func,argtype,resulttype
```

An argument is always specified; if no argument was specified in the Parallel Pascal program (e.g. using ``eof'' with no argument) the default (``input'' in this case) is explicitly specified by the compiler ``front-end''.

B.5.2 Transformational Functions

In addition to the standard functions provided by Pascal, Parallel Pascal contains some standard functions which perform transformations on entire arrays. This set of functions includes ``shift'', ``rotate'', ``trans'', ``expand'', and the reduction functions (``any'', ``all'', ``max'', ``min'', ``prod'', and ``sum''). In all of these cases the only argument to the function whose type varies is the array to be transformed. The stack is marked, the arguments are pushed, and the function is called. The old type of the array and the type of the function result are specified in the CSP instruction.

The ``shift'' and ``rotate'' functions have the following calling sequence:

```
MST  0
LLA  <array address>
IX?  ...  ;specify index information
LDC  integer,<#>    ;one for each array dimension
CSP  func,arrtype,resulttype
```

The ``expand'' function has the following calling sequence:

```
MST  0
LLA  <array address>
IX?  ...  ;specify index information
LDC  integer,<#>    ;new dimension
LDC  integer,<#>    ;low bound of new dimension
LDC  integer,<#>    ;high bound of new dimension
CSP  expand,arrtype,resulttype
```

The reduction functions have the following calling sequence:

```
MST  0
LLA  <array address>
IX?  ...  ;specify index information
LDC  integer,(<set of dimensions>)
CSP  func,arrtype,resulttype
```

Note that the dimensions along which the reduction is to take place are specified in one powerset constant.

B.5.3 Input and Output Procedures

The standard procedures ``get'', ``put'', ``reset'', and ``rewrite'' each operate upon one argument. The calling sequence is

```
LLA  <file>
CSP  func,filetype,nil
```

where ``filetype'' is the logical type of the argument.

The standard procedures ``read'' and ``write'' are actually implemented by several specialized procedures. When the file type is not ``text'' (that is, not file of char) the following equivalent sequences are used:

```
read(f,x) ≡    x := f↑; get(f)
write(f,x)     ≡    f↑ := x; put(f)
```

When the file is of type ``text'' the standard procedures ``rdi'', ``rdr'', and ``rdc'' are used for reading integers, real numbers, and characters (respectively); the standard procedures ``wri'', ``wrr'', ``wrc'', and ``wrs'' are used for writing integers, real numbers, characters, and strings (respectively).

The read functions have the calling sequence:

```
LLA  <file>
CSP  func,text,resultctype
```

where ``text'' is the type name for a file of char.

The write procedures require additional arguments. These specify the field width and the scale factor (this is meaningful only for floating-point numbers). The calling sequence for ``wri'' and ``wrc'' is:

```
<compute expression to be output>
LDC  integer,<width>
LLA  <file>
CSP  func,exprtype,nil
```

The ``wrr'' function requires the scale factor:

```
<compute expression to be output>
LDC  integer,<width>
LDC  integer,<scalefactor>
LLA  <file>
CSP  wrr,exprtype,nil
```

The ``wrs'' function requires one additional parameter - the string length:

```
LLA  <string>
LDC  integer,<width>
LDC  integer,<length>
LLA  <file>
CSP  wrs,stringtype,nil
```

B.5.4 Miscellaneous Standard Procedures

The procedures ``new'' and ``dispose'' are used for dynamic memory allocation. The calling sequence is:

```
LLA  <pointer>
CSP  func,pointertype,nil
```

There is no concept of packed data in Parallel P-code; hence, the Parallel Pascal procedures ``pack'' and ``unpack'' have no Parallel P-code counterparts.

B.6 Masking

Normally, all selected elements of arrays participate in all operations. A subset of these elements can be selected by specifying a mask. When a mask is in effect, all array assignments must conform to the shape of the mask. Masking is done with the use of a mask stack. If the stack is empty, no masking is in effect. If the stack is non-empty, the top of the mask stack specifies the current mask. The mask stack can be implemented as a set of pointers to values on the runtime (expression) stack. Expressions which are used to construct masks remain on the expression stack until the mask is removed. (They are never examined by the compiled code after they are calculated; thus, the storage specified by the runtime stack may be used to hold temporaries for the mask stack.)

A new mask is established with the WHR (``where'') instruction. The top of the expression stack is logically ANDed with the top of the mask stack, and the result is pushed on to the mask stack. (If there was no previous mask, the expression is simply pushed onto the mask stack; if there was a previous mask its type and the type of the new expression must be

identical.)

The OTW (``otherwise'') instruction provides the means for reversing the sense of the last conditional. If there is only one mask on the mask stack, it is complemented. Otherwise, the new mask is computed as the exclusive-or of the current mask (top of the mask stack) and the previous mask (next-to-top of the mask stack).

The ENW (``end where'') instruction is used to ``pop'' the mask stack. The mask stack and the runtime stack are popped. If the mask stack is now empty, the effect of masking is removed.

Masking only affects the STO and STR instructions (i.e. only assignments). The effect of a mask is not transmitted to any called procedures or functions.

APPENDIX C: HIGH LEVEL LANGUAGES FOR PARALLEL MATRIX PROCESSORS

A number of languages exist which could potentially be implemented on a parallel matrix processor. This appendix considers these languages. They can be grouped into conventional languages (that is, languages designed for conventional machines) and array languages (those designed with parallel processing in mind).

C.1 Conventional Languages

There is a wide variety of languages in this class. Of these languages, three stand out as possible candidates: APL (because of its inherent array capabilities), FORTRAN (because of its popularity among scientists and engineers), and PASCAL (because of its growing popularity in the programming community).

C.1.1 APL

APL (``A Programming Language'') was originally developed by Iverson as a mathematical notation. It is characterized by a rich set of primitive functions, compact notation, and flexible data handling. The type, shape, and size of data is runtime dependent, and primitive functions as well as properly-written

user functions can operate upon data of greatly diverse size.

APL provides a direct means for specifying parallel operations - entire arrays may be manipulated at once in a variety of ways. Unfortunately, the flexibility of the size and shape of expressions is often obtained at a high execution cost. APL implementations usually involve some degree of interpreted code (or periodic recompilation of code to adapt to new data shapes).

APL provides no data structures in addition to the (very flexible) array. The only control flow construct (aside from function calls, which may be recursive) is the branch statement. Some programmers dislike APL because it is by nature unconventional. These factors, in combination with the concern over its runtime efficiency make it unsuitable for direct implementation on a parallel matrix processor. Only a few restrictions need to be made to the APL language for it to be completely compilable. These restrictions and other modifications to the APL to make it suitable for parallel matrix processors are described in reference 1.

C.1.2 FORTRAN

FORTRAN is, in a sense, the ``grandfather'' of high-level languages. Designed in the early 1950's, it is the oldest high-level language still in use. It was designed for numerical computation (the name stands for ``FORmula TRANslation''), and many highly-optimizing compilers for FORTRAN produce very fast numerical code. Because of its age and dominance in the

scientific processing field it is familiar to most scientists and engineers. Its widespread use has also resulted in the development of a number of software libraries which assist in the construction of large programs.

FORTRAN's age is a mixed blessing, however. The language was developed before lexical analysis and parsing were fully understood, and its syntax is flawed in a number of ways. It is not conducive to structured programming (although the 1977 standard does provide a number of revisions toward this end). It provides no data structuring facilities other than the array, and is very awkward when dealing with character data or complex program control flow. It does not provide any aggregate array operations (although array facilities are under consideration for the next FORTRAN standard)[2]. It therefore is unattractive as a language for a parallel matrix processor.

C.1.3 Pascal

The programming language Pascal[3] was designed to achieve several goals, including[4]

- To make available a notation in which the fundamental concepts and structures of programming are expressible in a systematic, precise, and appropriate way.
- To make a notation available which takes into account the various new insights concerning systematic methods of program development.

- To demonstrate that a language with a rich set of flexible data and program structuring facilities can be implemented by an efficient and moderately sized compiler.

The resulting language has been the center of a great deal of attention since its development. It is making inroads into areas previously occupied by FORTRAN (for example, introductory programming courses at many universities) and has become very popular in the so-called ''personal computer'' market.

Pascal provides a flexible data structuring facility, permitting programmers to collect data into aggregate structures (records) and to define enumerated scalar types to provide mnemonic access to flag variables, etc. To reduce the errors which occur from incorrectly specifying the type of a data item, strong type checking is enforced. Type compatibility is checked at compile time whenever possible (thereby providing for fast execution).

Pascal is not without its faults. There has been some discussion concerning ambiguities in the typing mechanism and insecurities in the use of records[5,6,7]. Two other problems are particularly distressing: the lack of a separate compilation facility and the lack of dynamic-length arrays.

Some implementations of Pascal do provide for separate compilation (Wirth's PASCAL 6000, for example), but these often are done in a way which eliminates the advantages of Pascal's strong type checking. There have been a number of proposals

addressing this issue[8,9,10]. Perhaps the reason for this omission is the philosophy expressed by Wirth[4] that with a sufficiently fast compiler (and no linkage editor) it would be acceptable to make all changes at the source level, and merge sources together. Unfortunately, this philosophy does not work well on most systems where recompilation is expensive, especially when the change which forced the recompilation affects only a small portion of the code.

The fixed-size array problem results from Pascal's strong type checking and the fact that the array index ranges are considered part of the array type. This is especially limiting when an array is passed as a parameter to a procedure or function, prohibiting the design of ''library functions'' such as a general sort routine. A number of solutions have been suggested[11,12,13] including a parameterization scheme proposed by Wirth[14]. More recently, the ISO Pascal standard[15] has introduced the concept of a ''conformant array schema'', a means by which a parameter to a procedure or function may be an array whose index range is determined when the procedure or function is called.

Despite its limitations, Pascal is a powerful language which can be efficiently implemented. Although the standard language does not possess any facilities for expressing parallel computation, it forms an attractive base upon which such facilities could be built.

C.2 Array Languages

The languages discussed above were designed for efficient implementation on a conventional (non-parallel) processor. In order to efficiently execute programs on an SIMD-class parallel processor, it is necessary that computations be performed in parallel whenever possible. There are basically three ways to achieve this goal - using a vectorizing compiler, a language which directly specifies the implementation, or a language which directly specifies the parallelism but not the low-level implementation. In the following sections, each approach is considered.

C.2.1 Vectorizing Compilers

The first approach is to use a conventional language and write a compiler which can detect operations that can be performed in parallel. (The portion of the compiler which performs this task is often referred to as a ``vectorizer.'') Two examples of this technique are ILLIAC IV Fortran and the Paraphrase vectorizer.

C.2.1.1 ILLIAC IV Fortran

On the ILLIAC IV, the Fortran compiler contains a phase called the ``Paralyzer'' (for ``parallelism analyzer and synthesizer'') which performs parallelism analysis and converts the original Fortran code into IVTRAN, an extended Fortran dialect[16,17]. (IVTRAN is discussed in more detail in section

B.2.2.4.) The Paralyzer analyzes nested DO loops and extracts the inherent parallelism, subject to a number of restrictions. The Paralyzer output is then further processed by the IVTRAN compiler to produce the object program. Since the Paralyzer accepts standard Fortran as input, the use of the Paralyzer with IVTRAN permitted an ILLIAC IV user to run standard Fortran programs on the ILLIAC IV with no changes.

C.2.1.2 PARAPHRASE

The PARAPHRASE vectorizer[18] is not, by itself, a compiler. Rather, it was designed as a preprocessor for SIMD machines (the specific focus in this case was on pipelined vector machines). It performs a number of source-to-source optimizations on FORTRAN programs which restructure those programs for parallel execution.

PARAPHRASE produces output in standard Fortran with only two extensions - the specification of ''vector loops'' to mark loops which can be executed in parallel, and a provision for masking conditionals by a mode vector. As a result of this approach, the output from PARAPHRASE is relatively portable. This output can then be processed by a relatively unsophisticated compiler for the target machine to produce the final object code.

C.2.1.3 Vectorizing Compilers: Conclusions

The advantage of the vectorizing compiler approach is that the programmer need not learn anything new. Unfortunately, language systems based upon vectorizing compilers suffer from

several problems. One major problem is the vectorizer itself. In order to be able to extract a large degree of parallelism from an algorithm, the vectorizer will be complex. Like the vectorizers described above, most vectorizers operate upon nested iterative structures (e.g. nested FOR loops) and there are many special cases which can frustrate attempts to fully extract the parallelism that is present.

A more serious problem with the vectorizer approach is that it does not account for the different nature of the architecture upon which the program will be run. In many cases, algorithms which are optimal on a scalar (conventional) processor are not suitable for implementation on a parallel processor. In order to effectively program a parallel processor it is necessary to ''think parallel.'' A vectorizing compiler hides this fact from the user; thus, a programmer who uses such a compiler may be reluctant to change his programming practices, believing erroneously that the compiler will do as well as he would. These reasons discourage the use of a vectorizing compiler for a parallel matrix processor language.

C.2.2 Direct Specification of Implementation

The second approach to implementing a high-level language system is to design a language which fully exposes the architecture of the machine to the user. In a sense, the result is a ''high-level assembly language.'' The following sections describe (in alphabetical order) some languages in this category.

C.2.2.1 CFD

CFD is a Fortran dialect that was developed by the Computational Fluid Dynamics branch of NASA Ames Research Center for the ILLIAC IV[19]. It was designed for the applications area of fluid flow analysis for which programs which had previously been coded in standard FORTRAN.

CFD provides two forms of variables: CU (control unit) variables, which hold scalars, and PE (processing element) variables, which hold vectors. The first dimension of a PE variable is always 64 elements long and is represented by an asterisk. For instance, the following statements declare a 64-element vector and a square 64×64 matrix:

```
*PE INTEGER X(*)
*PE INTEGER MAT(*,64)
```

(The leading asterisk appears in all CFD statements except assignment statements.) Scalar variables are used as in standard FORTRAN. Array variables may be used as scalars (with one element selected) or as vectors of length 64 (that is, with every element along the first dimension selected). Hence, given the above definitions, the following would store the first column of ``MAT'' in ``X'':

```
X(*) = MAT(*,1)
```

Index arithmetic may be used to reposition data by circularly shifting it through the array; for instance,

```
X(*) = X(*+1)
```

rotates the vector ``X'' one position to the left. When the first subscript of an array is an asterisk, the second subscript (if any) may specify a vector expression. For instance:

```
MAT(*,X(*))    ≡    MAT(1,X(1)), MAT(2,X(2), ..., MAT(64,X(64))
```

Two provisions are made for selecting a subset of the 64-element vector. First, a parallel conditional statement may be used; for example,

```
*IF ((A(*).LT.0.)) A(*) = -A(*)
```

takes the absolute value of the vector A by storing only into those elements which are less than zero. Second, the 64 processors in the ILLIAC IV can be explicitly turned on or off by manipulating the logical vector ``MODE'':

```
MODE = (-A(*).LT.0.)
```

turns off all processors except those where the value of ``A'' is less than zero.

The special operators ``.ANY.'', ``.ALL.'', ``.NOT ANY.'', and ``.NOT ALL.'' can be used to construct scalar logical expressions from array logical expressions by performing the indicated operation (e.g. ``.ANY.'' returns ``.TRUE.'' if any element of its argument vector is true. The ``.SHL.'' and ``.SHR.'', and ``.RTL.'' and ``.RTR.'' operators perform left and right shifts and rotates (respectively) on bit vectors. Individual bits can be manipulated with the ``.TURN ON.'' and

``.TURN OFF.'' operators.

CFD also contains provisions for transferring data between the array memory and the main control unit. This is performed by the TRANSFER statement. For instance, the following statement transfers eight elements of the vector ``TEMP'' into the control unit array ``I'':

```
*TRANSFER (8) I=TEMP(1)
```

Although CFD permits the construction of extremely efficient programs, its heavy reliance on the structure of the the underlying machine (in this case, the ILLIAC IV), particularly in the number of processing elements and the vector nature of the machine, make it unattractive as the basis for a new language.

C.2.2.2 DAP Fortran

DAP Fortran is a Fortran dialect for the Distributed Array Processor[20]. The DAP was designed to be connected to a host computer as a memory module with internal processing capabilities. DAP Fortran reflects this design.

A complete program consists of a main program and set of subroutines written in standard Fortran for the host computer, along with a set of subroutines written in DAP Fortran. The host computer loads and starts the DAP; thereafter the two programs can operate asynchronously. Communication is carried out through a COMMON block. (The host processor can access the DAP as a

memory unit at all times, even when the DAP is processing data.)

DAP Fortran provides two basic data types: vectors and matrices. Arrays of higher orders (that is, with more dimensions) are represented as indexed sets of vectors or matrices. The size of a vector (or the dimensions of a matrix) must be the same as the hardware array size. The array dimensions are not explicitly stated when declaring the array; for example, the two-dimensional array ``A'' would be declared with:

```
REAL A(,)
```

Two different data representations are used for vectors and matrices; the representation is automatically changed when the language semantics call for it.

DAP Fortran permits elements in a vector or set to be indexed in several ways. First, a scalar index may be used as in standard Fortran. Second, an index may be omitted; if this is done the entire range of that index is selected. Third, the notation

```
A(*I)
```

may be used; this specifies that the element selected by ``I'' is to be expanded to fill the entire vector. These methods can be combined; for example, the expression:

```
A(,*I)
```

returns an array the size of ``A'', every column of which is the

same as ``A(,I)''.

DAP Fortran contains a number of useful facilities, particularly array indexing facilities. However, the underlying structure of the machine is evident (especially with respect to array declarations). Also, DAP Fortran contains no input-output facilities; instead, this is accomplished by the host processor. Finally, DAP Fortran does not remedy many of the problems associated with Fortran and its basic syntax. These factors discourage the use of DAP Fortran as the basis for a general parallel matrix processor language.

C.2.2.3 Glypnir

Glypnir was the first high-level language sucessfully implemented on the ILLIAC IV[21]. It is based upon Algol 60, with extensions to allow the programmer to explictly specify the parallelism of his algorithm.

Glypnir provides two major categories of variables: CU (control unit) variables which are single words, and PE (processing element) variables which are swords (64-word items). Vectors of words or swords may also be defined. There are no higher-order arrays. The statements:

```
CU INTEGER CI
CU REAL VECTOR Z[100]
PE REAL A
PE REAL VECTOR V[100]
```

declare the variable ``CI'' to be a scalar integer, ``Z'' to be a 100-element array of real, ``Z'' to be a sword of real (actually,

a 64-element vector of real), and ''V'' to be a 100-element vector of swords (actually, a 100×64 matrix). In addition to these types, the type ''BOOLEAN'' may be used to define 64-bit Boolean variables. These are stored in the scalar memory, and there is a correspondence between every processing element and every bit in the Boolean word.

PE variables are never indexed along the ''parallel'' dimension. An index expression for the non-parallel dimension may be a scalar expression or it may involve PE variables. For instance, if ''I'' is an integer sword with values (I_0 = 0, I_1 = 1, I_{63} = 63), than the expression ''Z[I+1]'' would reference the following components of ''Z'': (1,0), (2,1), (3,2), ..., (64,63). This is referred to as a slice.

Although Glypnir does not provide any means for indexing an individual member of a sword, it does provide a means for accessing the individual bits within each word. For instance, the expression:

```
A.[0:20] := A.[21:10] + 1
```

will cause the 10-bit field starting at bit 21 of A to be added to 1 and stored in the first 20 bits of A. (If A is a sword, this is done simultaneously for every word of the sword.) This allows for dense packing of the (limited) available main memory.

Glypnir provides a ''pointer'' data type for dynamic memory allocation. Blocks of words and blocks of swords may be allocated and deallocated. A pointer variable may be either a

simple variable or a sword of pointers. There are two types of pointers: those which can point anywhere in memory and those which can only point to locations within a given memory module.

Glypnir extends the Algol 60 control-flow constructs for parallel expressions. Conditionals may involve swords; if so, then an enable pattern is set during the execution of each ``arm'' of the conditional to mask the execution in each processing element. The iteration constructs are extended to swords as well - the processor continues to loop until the controlling expression is not satisfied in any array elements.

Glypnir provides for the declaration of subroutines; however, recursion is not permitted and all arguments are passed ``by value.'' Subroutine arguments may be words, swords, or slices, and subroutines may return either word or sword values.

The structure of Glypnir is very significantly influenced by the underlying hardware (the ILLIAC IV). The lack of an indexing mechanism along the parallel dimension makes it a highly machine-dependent language. This fact, coupled with its vector nature, make it unsuitable as the basis of a new language for the class of parallel matrix processors.

C.2.2.4 IVTRAN

IVTRAN is a Fortran compiler for the ILLIAC IV[22,17]. It was designed for use with a vectorizing preprocessor (described above), but it also contains some provisions for directly

specifying parallelism. The principal provision is the DO FOR ALL statement:

DO n FOR ALL $(i_1, i_2, \ldots, i_n)/s$

where i_1, i_2, ... are subscript variables and s specifies the range over which they will vary. (n is a statement number which defines the end of the loop.) Within the body of the DO FOR ALL statement the control indicies may only appear in assignments (or conditional assignments) and all index expressions must be of the form

I
or I + C
or I - C

where ``I'' is one of the controlling indices and ``C'' is an expression not depending upon any of the controlling indices.

IVTRAN also provides a syntax for specifying in detail the memory allocation for an array. Array dimensions may be skewed or aligned within ILLIAC processing elements, depending upon the nature of the problem.

Because the alignment of arrays places limitations upon the use of the Fortran EQUIVALENCE statement, two new declaration statements are provided. OVERLAP is used to overlap array allocations, thereby saving memory space, and DEFINE is used to define new arrays (with different index ranges) that correspond to previously-allocated arrays in a specified manner. Together, OVERLAP and DEFINE provide most of the functionality of the

EQUIVALENCE statement.

IVTRAN provides mechanisms for specifying parallelism within DO loops in a fairly machine-independent fashion. However, for efficient program construction the programmer must deal very closely with the ILLIAC IV architecture in the area of array declarations, particularly concerning the alignment or skewing of array dimensions across the (one-dimensional) array of processing elements. This strong coupling to the underlying architecture limits IVTRAN's suitability for implementation on a parallel matrix processor.

C.2.2.5 Direct Implementation Specification: Conclusions

A programmer who is familiar with the machine architecture can write extremely efficient programs in a language which directly specifies the low-level implementation. Unfortunately, languages designed for a specific machine are usually very non-portable. In addition, it is somewhat undesirable that programmers be concerned with the specific details of the hardware implementation.

A survey of users' experiences with the ILLIAC IV[23] indicated that while users preferred to be able to directly express the parallelism in their programs, the need to coerce their algorithms to fit the underlying machine structure (as the available languages, especially Glypnir and CFD, required them to do) was considered a drawback. These reasons discouraged the use of the direct specification of the low-level implementation in

the language selected for parallel matrix processors.

C.2.3 Direct Specification of Parallelism

The third approach to parallel language design is based upon this idea. Languages in this category permit the direct specification of parallel operations without requiring the programmer to be intimately acquainted with the underlying hardware.

In view of the criteria established above for a ''good'' programming language, a language which is designed according to this third philosophy (that is, one which permits specification of parallelism without forcing the programmer to specify the exact hardware implementation) is highly desirable. A number of languages in this category already exist. The following sections discuss these languages and their suitability to languages (in alphabetical order) and their suitability for implementation on a parallel matrix processor.

C.2.3.1 Actus

Actus[24] is a Pascal-based language suitable for scientific programming on a vector processor. The original target machine for Actus was the ILLIAC IV, but the language was designed to be independent of the hardware upon which it is implemented.

The design of Actus reflects the results of the survey of ILLIAC IV users mentioned above[23]. Perrott lists the following design criteria for Actus:

- The idiosyncracies of the hardware should be hidden from the user as much as possible.
- The user should be able to express the parallelism of the problem directly.
- The user should be able to think in terms of a varying rather than a fixed extent of parallel processing.
- Control of the parallel processing should be possible both explicitly and through the data, as applicable.
- The user should be able to indicate the minimum working set size of the database (in those cases where the database is larger than the size of the fast memory).

Actus supports most of the standard Pascal types (the most significant omission is the lack of variant records) along with some additional types (short integer, short real) that, when supported by the underlying hardware, provide more efficient memory utilization. Parallelism is achieved through the use of arrays - in an array declaration, one dimension may be declared to be parallel by replacing the standard Pascal subrange symbol ``..'' with a colon. For example,

```
var xxx: array [1:m, 1..n] of real
```

declares ``xxx'' to be a m×n array, wherethe first dimension may be accessed in parallel. It is important to note that the programmer is free to choose any size for the parallel dimension

- its size is not constrained by the underlying hardware.

Actus provides for the definition of index sets and parallel constants for indexing and initializing arrays. The syntax for both is similar:

```
const parconst = initial : (increment) final
index indexset = initial : (increment) final
```

The expression to the right of the ``='' generates the following (ordered) set of values:

initial, initial+increment, initial+2×increment, ..., final

While index sets are used for the parallel dimension of an array, Actus allows the use of a vector (one-dimensional array) as another index. For example, given the declarations

```
var  diag: array [1:100] of integer;
     para: array [1:100, 1..100] of integer;
```

the statements

```
diag := 1:100;
para[1:100, diag[1:100]] := 0
```

are effectively the same as

```
for i := 1 to 100 do
    diag[i] := i;

for j := 1 to 100 do
    para[j, diag[j]] := 0;
```

(where ``i'' and ``j'' are arbitrarily-chosen integer variables).

The operators shift and rotate are provided to align data in a parallel expression. shift performs an end-off shift, while

rotate performs a circular rotation. As an example, the following:

```
index first50 = 1:50;
var para: array [1:100] of integer;

para[first50] := para[first50] + para[first50 shift 50];
```

is equivalent to

```
for i := 1 to 50 do
    para[i] := para[i] + para[i+50];
```

When parallel variables or constants are used in an Actus statement, the extent of parallelism must be the same for all of the participants. The extent of parallelism encompasses both the size of the various items and the way that they are accessed; this excludes statements such as

```
a[1:10] := a[2:11]
```

Such a statement must be written

```
a[1:10] := a[1:10 shift 1]
```

so that the extent of parallelism is clear.

The smallest program unit over which the extent of parallelism cannot change is the assignment statement; however, some of the control flow statements also define an extent of parallelism. Once an extent of parallelism has been defined by such a statement, it is signified in the controlled statements by a sharp character (``#'').

Control statements which specify an extent of parallelism

include parallel version of the Pascal *while*, *if*, and *for* statements, the new *while any*, *while all*, *if any*, and *if all* statements, and the new *within* statement. The *within* statement merely defines the extent of parallelism - it has no other effect upon the program flow.

Finally, Actus addresses a common problem among parallel processors - lack of sufficient high-speed (``core'') memory, requiring some form of automatic buffering or virtual memory. It provides a syntax for specifying the minimum working set size for an array, so that automatic memory management won't ``swap out'' crucial data.

Actus is a very attractive language for vector processors. It satisfies most of the criteria stated at the beginning of this chapter for a ``good'' programming language. It is based upon a well-understood language (Pascal) and therefore is relatively easy for programmers to learn, it can be efficiently compiled, it does not force programmers to think at the low level of a particular machine architecture, and it encourages the development of well-structured programs. Unfortunately, Actus is tied very strongly to a vector architecture, making it unsuitable for matrix processors (*e.g.* Actus allows only one dimension of an array to be accessed in parallel). This restriction was addressed by Perrott in the language Actus Plus.

C.2.3.2 Actus Plus

Actus Plus[25] is a revision of the language Actus, eliminating the one-dimensional restrictions of the original language. Arrays may be declared with any number of parallel dimensions.

Actus Plus allows considerably greater flexibility in the use of index sets than Actus does. Index sets may consist of a consecutive or skipped range (as in Actus):

index indexset = 1:(2)99

a broken range:

index indexset = 1:10, 91:100;

an arbitrary range:

index indexset = 1, 3, 6, 9;

or a repeated range:

index indexset = 1*10, 2*5, 1*10;

Index sets may be combined with the operators ``+'' (union), ``*'' (intersection), and ``-'' (set difference). The rotate operator may also be used (as in Actus) to rotate the members of an index set; e.g. the following are equivalent:

1, 5, 3, 4 rotate 1 ≡ 5, 3, 4, 1

Index sets play a crucial role in the specification of parallel expressions. Actus Plus permits an expression to combine any two items, provided that their extents of parallelism

are the same. Thus, the following

```
var
    mat1: array [1:n, 1:m] of real;
    mat2: array [1:m, 1:n] of real;

    mat1[1:n,1:m] := mat1[1:n,1:m] * mat2[1:m,1:n];
```

is equivalent to the Pascal code (where ``i'' and ``j'' are arbitrarily-chosen integer variables):

```
for i := 1 to n do
    for j := 1 to m do
        mat1[i,j] := mat1[i,j] * mat2[i,j];
```

Although the meaning of the above is clear the following similar case is ambiguous:

```
var
    row: array [1:n] of real;
    mat: array [1:n,1:n] of real;

    mat[1:n, 1:n] := mat[1:n,1:n] * row[1:n];
```

because it is not clear whether the multiplication should be performed along the rows or the columns of ``mat''. The ambiguity is resolved by using an index set:

```
index iset = 1:n;

    mat[1:n,iset] := mat[1:n,iset] * row[iset];    (* row mult *)
    mat[iset,1:n] := mat[iset,1:n] * row[iset];    (* column mult *)
```

The while, if, and case statements in Actus are also available in Actus Plus. Since these control constructs affect the extent of parallelism, a sharp-sign notation (similar to that in Actus) is used to represent the actual extent:

```
if a[1:n,1:m] <> 0 then
    a[#1,#2] := a[#1,#2] + 1;
```

Actus Plus is an attractive language. It provides for the direct specification of parallelism without forcing the programmer to know the detailed architecture of the machine on which his programs will run. The generalized index sets, and the flexible operators provided for manipulating them could be somewhat expensive to implement on a machine with a limited interconnection network. Nonetheless, Actus Plus would be a strong candidate for implementation on a parallel matrix processor. It did not influence the design of Parallel Pascal because it was not specified in time; in addition, no research results from an implementation of Actus Plus were available.

C.2.3.3 Proposed Extensions to ALA

The language ALA was proposed by Zosel as an extension to ALGOL for the STAR-100[26]. The language reflects the philosophies of APL and ALGOL-68.

Vector extensions to ALGOL are implemented in a natural way: a vector may be used wherever a scalar may be used, provided that there is an obvious interpretation of its meaning. Operands in an arithmetic expression must be conformable: either they must be the same size or one must be a scalar. The set of primitive data types includes all of the data types defined by the hardware, including 32- 64- and 128-bit floating point representations (on the STAR-100), the various integer representations, and bit

strings.

Control statements (e.g. conditional statements and loops) must have scalar control variables; however, the functions ``allof'' and ``anyof'' are provided to reduce array expressions to simple Boolean values.

ALA provides user-accessible descriptors for manipulating vectors. A descriptor ``DESC'' may be associated with a vector ``VECT'' by one of the following two statements:

```
DESC => VECT
DESC => VECT[slice]
```

In the first form, the descriptor refers to the entire ``VECT'' array; in the second, it refers only to a subset of the elements in ``VECT'' (the subset is determined by the ``slice''; the format of the ``slice'' is defined below). During the course of execution, the size of the vector referred to by the descriptor may change; however, this change will not be reflected in the original array. Descriptors are also used when an entire array is passed as a parameter to a subroutine. Rather than passing the array, the called routine receives a descriptor for the array.

ALA permits indexing by a scalar, a Boolean set, a set, a sparse set (a special STAR-100 capability), or a ``slice.'' A slice may have one of two forms:

```
I:J
I;J
```

In the first case, items ''I'' through ''J'' are selected. In the second case, all items except the first ''I'' and the last ''J'' are selected. All forms of indexing are valid on both sides of an assignment statement, and indexing may be performed on expressions as well as simple variables.

ALA has some desirable features; in particular, the ability to deal with vectors in the same fashion as (and in combination with) scalars is very appealing. However, ALA is very heavily weighted toward implementation on the STAR-100, and it includes features which may be expensive to provide on other machines. These include Boolean indexing, the use of sparse vectors, and the large runtime variability of the size of vectors. Also, a language designed for a vector processor is dissimilar in many ways from a language for a matrix processor.

C.2.3.4 APLISP

APLISP[27] is a language for image and speech processing. Its target machine is the partitionable SIMD/MIMD system PASM[28] but the language is machine independent.

The syntax of APLISP is similar in many ways to that of Pascal. Deviations from Pascal include the definition of two new fundamental data types (BYTE and INDEX), a flexible array indexing scheme, and conditional control statements.

Arrays in APLISP are viewed as a set of named objects, each of which is an ordered n-tuple consisting of the index (or

indices) and the value. (As an example, for a one-dimensional array, the objects are ordered pairs (i,x) where i is the index and x is the corresponding value.) Index sets are used to select subsets of these n-tuples. For multi-dimensional arrays, sets of index n-tuples may be specified by a Cartesian product or concatenation of two index sets.

Index sets may be used in assignment statements on both sides of the expression. Index sets which appear on both sides of an assignment are forced to correspond to one another; hence, the assignment A[U] := B[U] implies that for each u U, A[u] := B[u].

APLISP provides the WHERE statement for parallel conditional evaluation. Execution is controlled by a conditional expression over an index set. Within the body of the WHERE clause and optional ELSEWHERE clause the range of the index set is restricted to only those n-tuples for which the conditional is true or false, respectively.

APLISP provides a flexible mechanism for expressing parallelism without consideration of the underlying machine. The concept of index sets is a very powerful one (although as with APL, the uninitiated may object to the concise and highly-symbolic format). The runtime-dynamic shape and configuration of the index sets may pose an implementation problem on processors with restricted interconnect networks (e.g. matrix processors with simple near-neighbor connections). Nonetheless, APLISP has

many attractive features. It did not influence the design of Parallel Pascal because it was not specified in time; in addition, no research results from an implementation are available.

C.2.3.5 Fortran 8X

At the present time, the X3J3 committee of the American National Standards Institute is considering proposals for extensions to Fortran[2,29]. All of the proposed changes are still subject to change, so it is impossible at this time to determine the form of the new language (often referred to as "Fortran 8X"). However, it is instructive to consider some of the proposed extensions in the realm of array indexing and parallel processing.

The current proposals permit the use of unsubscripted array names in arithmetic expressions on either side of the assignment symbol (``=''). The evaluation and assignment is considered to be simultaneous for all array elements. (This definition facilitates the implementation on a parallel processor, where the evaluation and assignment is simultaneous, as well as on a conventional serial machine.)

When subscripts are specified, the special symbol ``*'' is used to represent the entire range of the array. For example:

```
A(1,*)          - select row 1
A(*,1)          - select column 1
A(-*,1)    - select column 1 in reverse order
```

Finally, an array section may be specified by a doublet or triplet:

```
V(1:5)          - select V(1), V(2), V(3), V(4), V(5)
V(1:K)          - select V(1), V(2), ... , V(K)
V(1:5:2)  - select V(1), V(3), V(5)     [step by 2]
```

Other capabilities which are under consideration are more complicated array sections, vector indexing for arrays, an IDENTIFY statement (to restrict the number of array elements which are active when an explicit index expression is not given), and a conditional assignment (WHERE) statement.

The proposals for Fortran 8X are of interest, because Fortran is one of the most widely-used high level languages in the field of scientific computing. However, it would be unwise to adopt the current proposals for Fortran 8X as the basis for a new language at this time, since it is likely that there will still be significant revisions to the language before a standard is adopted. Until some of the other problems with Fortran can be satisfactorily resolved (for example, its lack of facilities for structured programming), a language based upon Fortran and the proposals outlined above is not suitable for implementation on a parallel matrix processor.

C.2.3.6 Parallel Extensions to LRLTRAN (Fortran)

LRLTRAN is an extended Fortran in use at the Lawrence Livermore Laboratories in California. To accomodate the STAR-100, a number of vector extensions were made to the language[30].

The resulting language provides for the specification of efficient manipulation of vector quantities.

The one-dimensional nature of LRLTRAN is very apparent in the declaration of parallel variables. These variables are explicitly declared as vectors, e.g.

```
VECTOR A(99)
```

declares that ``A'' is a vector with 100 elements (the lower-bound of a vector index is always zero). Vectors so defined may be used in arithmetic expressions (with the expected results), e.g.

```
A = A + 1
```

increments each element of the vector ``A'' by 1.

In addition to declaring vector storage, one may also define vector descriptors:

```
VECTOR (BPTR,B)
```

In this case, the variable ``BPTR'' is a user-accessible description of the address and size of a vector. After setting ``BPTR'' appropriately, the desired data may be accessed as a vector via the descriptor ``B''. For example, if ``BPTR'' points to a 7-word area beginning at address 1000, then the statement

```
B = 1
```

will set words 1000..1006 to 1. Operators are provided to convert scalars to vector descriptors and vice versa and to

determine the length of a vector.

Vectors (or vector descriptors) which participate in assignment statements need not be the same size. Scalars are automatically extended to vectors during an assignment. If both the left- and right-sides of the assignment are vectors, the right-hand-side is completely evaluated and the results are assigned one-by-one until one of the two vectors is exhaused.

Vectors may be subscripted with a scalar, a vector, a contiguous range of elements, or a set (bit-vector); they may also be treated as sparse vectors.

Finally, LRLTRAN contains a number of intrinsic functions to provide for summing along vectors, merging vectors, etc. The implementation of these is somewhat unfortunate: unlike normal Fortran intrinsic functions, the user cannot override the standard definitions with his own. Even if he supplies a function definition the compiler will use the predefined intrinsic function.

LRLTRAN is a flexible language for dealing with the STAR-100, but because of its strictly-vector nature it is not well suited for a matrix processor. Many of its facilities (such as sparse vectors and vector indexing) are directly related to the hardware capabilities of the STAR-100 and may be very expensive on a processor with a more rigid structure.

C.2.3.7 PascalPL

PascalPL[31] is a Pascal-based language which facilitates parallel image processing. Its design was influenced by the architectures of contemporary parallel arrays. It is presently available at the University of Wisconsin, Madison, as a translator which converts PascalPL programs to standard Pascal programs.

A PascalPL program consists of a standard Pascal program which contains parallel procedures. The parallel procedures themselves may contain a mix of standard Pascal statements and parallel constructs. All parallel constructs are distinguished by the presence of two leading vertical lines (``||'', the standard symbol for parallelism).

A parallel procedure is introduced with the declaration

||procedure procedurename ;

At some point after this (between which there may be standard Pascal statements), a ``dimension declaration'' must be placed to define the bounds over which operations take place:

||dim [0..127, 0..127] ;

Optional fields also declare the data type (either integer - which is the default - or Boolean), the association of arrays with set names (sets of arrays), and the index mapping from the input array set to the result array set. Once the dimension has been defined, the following parallel constructs may be intermixed

with standard Pascal statements:

```
||read(...) ;
||write(...) ;
||set arrays_assigned_to := compound_of_arrays ;
||if compound_of_arrays inequality
  ||then arrays_modified
  ||else arrays_modified_on_failure ;    [optional]
||border := bordertype ;
```

The ||read and ||write constructs perform input and output of arrays or subsets of arrays. The ||border statement defines the value that is to be used when array indexing lies outside the declared array dimensions.

The most significant feature of PascalPL is its array indexing mechanisms. Array operations are performed by the ||set and ||if constructs. The ||set instruction unconditionally performs array assignments. The left-hand-side of the assignment specifies one or more result arrays; the right-hand-side specifies an array expression. The array expression may contain scalar variables (always preceeded in this context by a ``#'' character), constants, and array index expressions. A simple example is:

```
||set array1 := 2*array2 + array3 - #mean;
```

which performs (element-by-element) addition of 2 times ``array2'' with ``array3'', subtracts from each element the value of the scalar variable ``mean'', and stores the result (element by element) in ``array1''.

PascalPL provides even greater flexibility in array

expressions by permitting the specification of neighborhood operations. A specified set of near-neighbors may be individually weighted and combined by a specified operation. For instance, suppose that it is desired to compute

```
b[i,j] := a[i-1,j-1] + a[i-1,j+1] + a[i+1,j-1] + a[i+1,j+1] + 4*a[i,j
```

for all elements of ``a'' and ``b''. This can be accomplished with the following PascalPL statement:

```
||set b := a[+(-1:-1, -1:1, 1:-1, 1:1, 0:0*4)];
```

This mechanism is generalized even further to permit thresholding; e.g. the following performs the same sum, but only includes the value of the center if it is greater than 32:

```
||set b := a[+(-1:-1, -1:1, 1:-1, 1:1, 0:0*4>32)];
```

The conditional structure (||if) can be used to perform simple modifications to arrays on an element-by-element basis, as determined by the controlling conditional. For instance, to triple the value of all elements in the array ``b'' if the corresponding elements in array ``a'' are non-zero, the statement would be:

```
||if a <> 0
  ||then b*3;
```

PascalPL is a very intriguing language. It has a compact and powerful notation which is capable of representing many desirable operations which can be efficiently performed by a parallel matrix processor. The language appears to be easily

extensible to arrays with any number of dimensions. The ability to mix PascalPL and standard Pascal within the same program is also a definite advantage. Its biggest drawback may be its biggest feature - the symbolism used to express parallel operations. The operations that are specified for arrays are syntactically and semantically different than similar operations specified in standard Pascal. The conditional statement also operates upon arrays with a different syntax; multiplying an array ``xyz'' by 3 is done by

```
xyz*3
```

in an ||if statement, but by

```
||set xyz := xyz*3;
```

in an assignment statement. Thus, while PascalPL is a viable candidate for a parallel matrix processor such as the MPP, it seems desirable that a different approach, one which does not significantly distinguish between parallel and scalar operations, be taken.

C.2.3.8 VECTRAN

VECTRAN was proposed by researchers at IBM as an extension to IBM FORTRAN IV[32]. It was designed as an upward compatible extension to FORTRAN (66) for scientific applications programming.

The declaration statements RANGE and IDENTIFY are used to specify the array elements which participate in an operation.

The RANGE statement can be used to restrict the range of a parallel operation to some subset of the array; for example:

```
RANGE /N,M/ A(10,10), B(15,25)
...
N = 5
M = N+2
...
A = 2.5*A + B
```

Only the 5×7 subarrays of A and B participate in the parallel computation.

The IDENTIFY statement permits the redefinition of axes in the array, so that well-defined substructures of an array may be defined and used. For instance, the elements along the diagonal of a two-dimensional array may be ``identified'' with the elements of a one-dimensional array of the appropriate size.

Vector indexing is permitted in VECTRAN. The semantics are similar to APL. The order of the values in the vector is significant.

Parallel conditional control is provided by the WHEN and AT statements. These statements differ in the order in which evaluation is performed. WHEN fully evaluates the conditional expression and each controlled expression, and then performs conditional assignment. AT evaluates the conditional expression and then conditionally evaluates the appropriate controlled expression. Hence, WHEN performs conditional assignment while AT

performs conditional evaluation.

VECTRAN also provides functions for manipulating data logically (PACK and UNPACK) and arithmetically (e.g. matrix multiply).

VECTRAN has a number of desirable features. It is based upon a well-known language (Fortran, albeit Fortran 66), it provides for flexible manipulation of data, and it is targeted toward numerical applications. Unfortunately, VECTRAN does not remedy many of the problems in Fortran - poor control flow constructs, the lack of user-defined data types with their associated type checking, and Fortran's generally poor syntax. Also, its indexing mechanisms, particularly vector indexing, are complex to implement; this may seriously impact the efficiency of an implementation on a parallel matrix processor.

C.2.3.9 Direct Parallelism Specification: Conclusions

A language which permits the direct specification of parallelism, without requiring the specification of the low-level implementation, is very attractive. However, none of the languages described above (with the exception of Actus Plus, whose full description was unknown at the time of the initial language survey) is entirely suitable for implementation on a parallel matrix processor such as the MPP. The languages are either too vector-oriented or too general for efficient implementation.

C.3 References

1 Anthony P. Reeves, John D. Bruner, and Tony M. Brewer, "High Level Languages for the Massively Parallel Processor," TR-EE 81-45, School of Electrical Engineering, Purdue University, West Lafayette, IN (November 1981).

2 Loren P. Meissner, "The Fortran Programming Language - Recent Developments and a View of the Future," ACM FORTEC FORUM Vol. 1(1), pp.3-8 (July 1982).

3 Kathleen Jensen and Niklaus Wirth, PASCAL User Manual and Report, Springer-Verlag, Berlin, Heidelberg, New York (1974).

4 Niklaus Wirth, "The Design of a PASCAL Compiler," Software - Practice and Experience Vol. 1, pp.309-333 (1971).

5 M. Iglewski and J. Madey, "A Contribution to an Improvement of Pascal," ACM SIGPLAN Notices(1), pp.48-58 (January 1978).

6 R. D. Tennent, "Another Look at Type Compatibility in Pascal," Software - Practice and Experience Vol. 8, pp.429-437 (1978).

7 J. Welsh, W. J. Sneeringer, and C. A. R. Hoare, "Ambiguities and Insecurities in Pascal," Software - Practice and Experience Vol. 7, pp.685-696 (1977).

8 Augusto Celentano, Pierluigi Della Vigna, Carlo Ghezzi, and Dino Mandrioli, "Separate Compilation and Partial Specification in Pascal," IEEE Transactions on Software Engineering Vol. SE-6(4), pp.320-328 (July 1980).

9 Richard J. LeBlanc and Charles N. Fischer, "On Implementing Separate Compilation In Block-Structured Languages," ACM SIGPLAN Notices Vol. 14(8), pp.139-143 (August 1979).

10 Richard J. LeBlanc, "Extensions to PASCAL for Separate Compilation," ACM SIGPLAN Notices Vol. 13(9), pp.30-33 (September 1978).

11 Edward N. Kittlitz, "Another Proposal for Variable Size Arrays in PASCAL," ACM SIGPLAN Notices Vol. 12(1), pp.82-86 (January 1977).

12 Sergei Pokrovsky, "Formal Types and Their Application to Dynamic Arrays in Pascal," ACM SIGPLAN Notices, pp.36-42 (October 1976).

13 B. J. MacLennan, "A Note on Dynamic Arrays in PASCAL," ACM SIGPLAN Notices, pp.39-40 (September 1975).

14 Niklaus Wirth, "An Assessment of the Programming Language Pascal," IEEE Transactions on Software Engineering Vol. SE-1(2), pp.192-198 (June 1975).

15 A. M. Addyman, "A Draft Proposal for Pascal," ACM SIGPLAN Notices Vol. 15(4), pp.1-66 (April 1980).

16 David L. Presberg, "The Paralyzer: IVTRAN's Parallelism Analyzer and Synthesizer," ACM SIGPLAN Notices Vol. 10(3), pp.9-16 (March 1975).

17 Robert E. Millstein, "The ILLIAC IV Fortran Compiler," ACM SIGPLAN Notices Vol. 10(3), pp.1-8 (March 1975).

18 D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," COMPSAC, pp.709-715 (1980).

19 K. G. Stevens, Jr., "CFD - A FORTRAN-like Language for the ILLIAC IV," ACM SIGPLAN Notices Vol. 10(3), pp.72-75 (March 1975).

20 P. M. Flanders, D. J. Hunt, S. F. Reddaway, and D. Parkinson, "Efficient High Speed Computing with the Distributed Array Processor," pp. 113-127 in High Speed Computer and Algorithm Organization, ed. David J. Kuck, Duncan H. Lawrie, Ahmed H. Sameh, Academic Press (1977).

21 D. H. Lawrie, T. Layman, D. Baer, and J. M. Randal, "Glypnir - A Programming Language for Illiac IV," Communications of the ACM Vol. 18(3), pp.157-164 (March 1975).

22 David B. Erickson, "Array Processing on an Array Processor," ACM SIGPLAN Notices Vol. 10(3), pp.17-24 (March 1975).

23 R. H. Perrott and D. K. Stevenson, "Users' Experiences with the ILLIAC IV System and its Programming Languages," ACM SIGPLAN Notices Vol. 16(7), pp.75-88 (July 1981).

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

25 R. H. Perrott, "Actus Plus," CS 008, Department of Computer Science, The Queen's University, Belfast, Ireland (November 1981).

26 Mary E. Zosel, "A Modest Proposal for Vector Extensions to ALGOL," ACM SIGPLAN Notices Vol. 10(3), pp.62-71 (March 1975).

27 Philip T. Mueller, Leah J. Siegel, and Howard Jay Siegel, "A Parallel Language for Image and Speech Processing,"

Proceedings, COMPSAC 80, pp.476-483 (October 1980).

28 Howard J. Siegel, Leah J. Siegel, Frederick C. Kemmerer, Philip T. Mueller, Jr., Harold E. Smalley, and S. Diane Smith, "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition," IEEE Transactions on Computers Vol. C-30(12), pp.934-947 (December 1981).

29 Richard R. Ragan, "Proposed Argument Association Rules for Arrays in FORTRAN 8x," ACM FORTEC FORUM Vol. 1(1), pp.9-14 (July 1982).

30 R. G. Zwakenberg, "Vector Extensions to LRLTRAN," ACM SIGPLAN Notices Vol. 10(3), pp.77-86 (March 1975).

31 Leonard Uhr, "A Language for Parallel Processing of Arrays, Embedded in Pascal," pp. 53-87 in Languages and Architectures for Image Processing, ed. M.J.B. Duff and S. Levialdi, Academic Press, London (1981).

32 George Paul and M. Wayne Wilson, "An Introduction to VECTRAN and Its Use in Scientific Applications Programming," 7287 (#31383), IBM T. J. Watson Research Center, Yorktown Heights, NY (September 1978).