Allocation and Manipulation of Records in the NON-VON Supercomputer[1]

CUCS-211-83

David Elliot Shaw
and
Bruce K. Hillyer

Department of Computer Science
Columbia University

July, 1983

Abstract

NON-VON is a highly parallel supercomputer, portions of which are now under construction at Columbia University. A full-scale NON-VON prototype might comprise as many as a million tiny processing elements, each associated with a small random access memory. Among the principal goals of the NON-VON Project is the development of programming languages and compilers that realize the machine's potential for massive parallelism while insulating the user from the details of its tree-structured physical topology.

One conceptual metaphor that has proven useful in pursuing this goal is the notion of an intelligent record, a primitive data element of arbitrary size that functions as if it were associated with its own dedicated computer. This paper describes the essential mechanisms used to support intelligent records within a high-level parallel programming language environment. We then illustrate the use of these techniques in a few simple applications and explore certain time/space tradeoffs that characterize alternative record allocation schemes.

1 Introduction

Over the past few years, growing numbers of computer scientists have become interested in alternatives to the conventional "von Neumann" computer. Computer architects see new opportunities in the the emerging possibilities

---

[1]This research was supported in part by the Defense Advanced Research Projects Agency under contract N00039-82-C-0427.

for large-scale parallelism afforded by the technology of very large scale integrated (VLSI) circuits. Language designers are exploring alternatives to the von Neumann model of computation for the description of computational tasks.

A particularly forceful case against the von Neumann machine has been made by Dr. John Backus in his influential 1977 Turing Award lecture, entitled "Can Programming be Liberated from the von Neumann Style?" [Backus, 1978]. Among the central themes of this lecture was the contention that our preoccupation with conventional computer hardware, whose operation is based on serial, "word-at-a-time" transfers between memory and CPU through what Backus calls the "von Neumann bottleneck", has had an unfortunate affect on the evolution of programming languages. Research related to the NON-VON machine architecture [Shaw, 1982] seeks to advance beyond the hardware and software limitations of the "von Neumann bottleneck".

Implemented using nMOS VLSI circuits custom-designed at Columbia and fabricated using DARPA's MOSIS system, NON-VON is one of the most ambitious attempts to date to realize very large scale parallelism using current integrated circuit technology. A full-scale NON-VON machine might incorporate between 100,000 and 1,000,000 tiny processors, each associated with a small (on the order of 64 bytes) random access memory. In the NON-VON architecture, a number of these unusually area-efficient processing elements are embodied within each integrated circuit. The individual processing elements are interconnected in such a way as to allow them to simultaneously perform useful computational work, supporting an effective processor-memory bandwidth many orders of magnitude in excess of today's fastest machines.

While the realization of massive parallelism is central to the goals of NON-VON Project, it must be emphasized that the machine has been designed as much for ease of programming as for the attainment of unprecedented execution speed. Much of our attention has thus been directed toward the design and compilation of high-level languages capable of fully exploiting NON-VON's potential parallelism while insulating the user from the low-level details of its physical organization.

Intuition might suggest that the use of such a parallel programming language would be far more difficult than the generation of software for a conventional machine. It has been our experience, however, that a large share of the operations that are performed in practice are in fact more easily described in such languages than in those typically used to program conventional machines. To be sure, the highly nonstandard organization of the NON-VON machine raises a number of new issues for the language designer. On the other hand, NON-VON's architecture obviates the need for many of the loops, indexes, pointers, hashing techniques, and other constructs found in conventional applications software that are in fact extrinsic to the semantics of the problem domain.

This paper focuses on some of the most important mechanisms used in the implementation of very high level linguistic constructs for NON-VON. In the following section, we briefly describe the physical organization of the NON-

VON machine, and outline the essential capabilities of its instruction set. The remainder of the paper is dedicated to software- and language-related issues. First, we discuss the use of abstraction to manage the complexity of NON-VON software, and introduce a simple extension of Pascal capable of exploiting the machine's potential for massive parallelism. We then provide examples of algorithms and code, at both a low level (to exemplify the detailed behavior of the machine) and a high level (to illustrate the management of complexity). Considerable attention is given to the means by which the physical mapping of logical records onto physical processing elements is made transparent to the programmer, while preserving the full potential parallelism of the machine.

2 The NON-VON Supercomputer

Previous supercomputer designs have tended to focus on the efficient solution of a rather specialized class of numerical problems. NON-VON, on the other hand, is intended to provide highly efficient support for a wide range of numerical and non-numerical applications involving the manipulation of large quantities of data. In particular, highly efficient support is provided for the kinds of operations which seem to characterize much of the workload involved in commercial database management and data processing applications.

The NON-VON architecture comprises three subsystems: the Primary Processing Subsystem (PPS), the Secondary Processing Subsystem (SPS) and the Control Processor (CP). Briefly, the PPS incorporates a large number of simple, highly area-efficient Processing Elements (PE's), configured as a binary tree. The SPS is based on a number of "intelligent disks" whose individual disk heads are each associated with a small amount of hardware capable of dynamically examining the data that pass underneath them, and passing selected records along to the PPS in a highly parallel fashion. The CP is a conventional general purpose computer that broadcasts instructions to be executed simultaneously by all PE's in the PPS.

A brief overview of the structure and function of the machine is presented below. Readers familiar with the NON-VON architecture may wish to skip to the beginning of the following section.

2.1 Primary Processing Subsystem

NON-VON's Primary Processing Subsystem embodies a vast number (on the order of a million, during the intended time frame for practical application of the machine) of very simple processing elements, each containing its own locally accessible random access memory. Although the instruction set of an individual PE can support up to 256 bytes of RAM, a current realization of the architecture might incorporate on the order of 64 bytes of local storage per

processing element. The processor data path, communication circuitry and control logic together occupies only about as much silicon area as the local RAM, allowing a number of PE's to be embedded on a single chip. (During the target time frame, each chip might contain between 64 and 128 such processing elements.) Each PE contains an I/O switch, which is used to support several modes of inter-processor communication.

In practice, many or all of these tiny PE's are often able to operate concurrently on data stored in their respective local memories, supporting effective execution speeds far exceeding those of today's fastest supercomputers. Because of their small size, however, the PPS is expected to be scarcely more expensive than an equivalent amount of ordinary random access memory. From the viewpoint of performance, the PPS may thus be regarded as an ultra-high-speed parallel processing ensemble; from a cost perspective, though, it is better viewed as a (slightly overpriced) random access memory unit.

The physical organization of the PPS is based on a scheme proposed by Leiserson [1981], in which a single type of fully "scalable" chip is used to build a tree of arbitrary size. Each chip contains a complete binary subtree and a single internal node. The number of PE's in the subtree may be increased arbitrarily as device dimensions continue to decrease without increasing the number of pins per package. As illustrated in Figure 1, the Leiserson packaging scheme has the property that two such chips may be interconnected to form a larger tree having precisely the same external connections as a single chip. This process may be repeated recursively to construct an arbitrarily large tree using printed circuit boards having a regular, planar, area-efficient layout. Connections between boards, and (in a large machine) between cabinets, are made in the same manner as those between chips. Unlike most other interconnection topologies proposed for use in highly parallel machines [Kung and Leiserson, 1979; Leighton, 1981], the number of wires required to interconnect the various system modules is thus independent of the size of the PPS, allowing the construction of very large machines having manageable inter-modular wiring.

The cornerstone of the NON-VON architecture is the association of a small amount of RAM (at most 256 bytes) with each of a vast number of simple processors. The processors are physically interconnected with a communication pattern in the form of a binary tree. The hardware also provides adjacent neighbor communication, with neighbors defined by adjacency in an inorder traversal of that tree. The processor - memory pairs, referred to as processing elements (PE's), cooperate in single instruction stream, multiple data stream (SIMD) mode, under the direction of a general purpose computer attached to the root of the tree of PE's. This control processor (CP) is responsible for the broadcast of instructions through the root and down into the tree, where the instructions are executed in parallel by the PE's.

In addition to the tree neighbor communication established by the topology of the physical interconnections, the I/O switches in each NON-VON PE provide communication between linearly adjacent neighbors, which are defined to be PE's that are consecutive in an inorder traversal of the tree. All PE's can

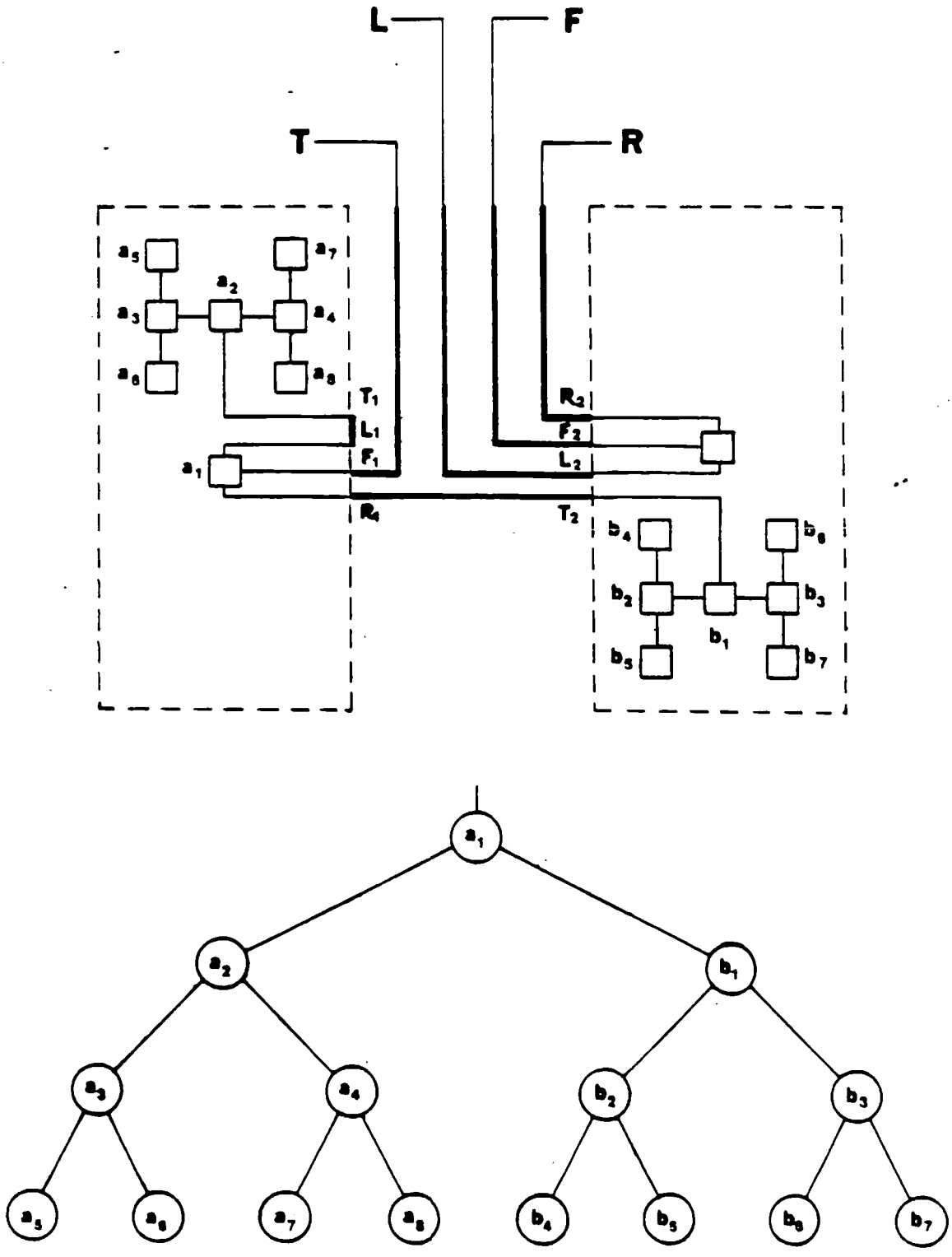

Figure 1: Interconnection of Two Leiserson Chips

receive data in parallel from their parent (or left child, or right child). Parallel communication with left neighbors, or right neighbors may also be performed, although two time steps are required, as illustrated by the next figure.

Broadcast communication is also supported by NON-VON. A byte may be sent to all PE's simultaneously, reaching the leaves of the tree of PE's through combinational logic within one instruction time. One other communication pattern supported by the I/O switch design is not a parallel operation: a uniquely enabled PE may be instructed to report a byte of data, which will be propagated up the tree and out through the root with a small combinational logic delay.

The first version of the NON-VON PE, which we have come to call NON-VON 1, has an architecture comprising four functional units. The first is an I/O switch handling 8-bit and 1-bit communication paths among the PE, its children, and parent. The second portion is a PLA that receives incoming 8-bit instructions from the I/O switch, and generates appropriate control signals for the PE's logic. The third part is a 64 byte RAM. The fourth unit contains the processing power. It is organized around two data buses, one for byte-wide data, and the other for 1-bit data. There are eight byte-wide registers, including a memory address register (MAR), an I/O register (IO8), and two accumulators (A8, B8). The eight 1-bit flag registers include the enable register (EN1), an I/O register (IO1), and two 1-bit accumulators (A1, B1), as well as a carry bit (C1). There is a comparison unit that compares the contents of A8 and B8, sets A1 if A8 = B8, and sets B1 if A8 > B8. The 1-bit ALU has both arithmetic and logical functions. The arithmetic implemented is 1-bit add with carry, and 1-bit subtract with borrow, using operands from A1 and B1 as well as the carry bit from C1. The result is left in A1 and C1. All 16 Boolean functions of A1 and B1 can be performed, with the result developed in A1.

The instruction set for NON-VON 1 is fairly ordinary in many respects. There are no branch instructions, however, as programs are not stored locally. The only addressing modes for RAM are absolute, and indirect through the memory address register, which can be manipulated by the instruction set. There are also some special instructions for NON-VON, as described below. The instruction set consists of six groups:

1. Register Transfer Group: NON-VON 1 can load and store A1, A8, B1, and B8 from any of the eight registers having the correct length.

2. Memory Access Group: Transfers between A8 and RAM may be performed, with the address found either in the MAR or as the second byte of the instruction.

3. Arithmetic and Shift Group: One-bit add and subtract are implemented, as described previously. Additionally, there are bi-directional 9-bit circular shifts of the A and B registers, with A8+A1 forming one loop, and B8+B1 forming the other. These

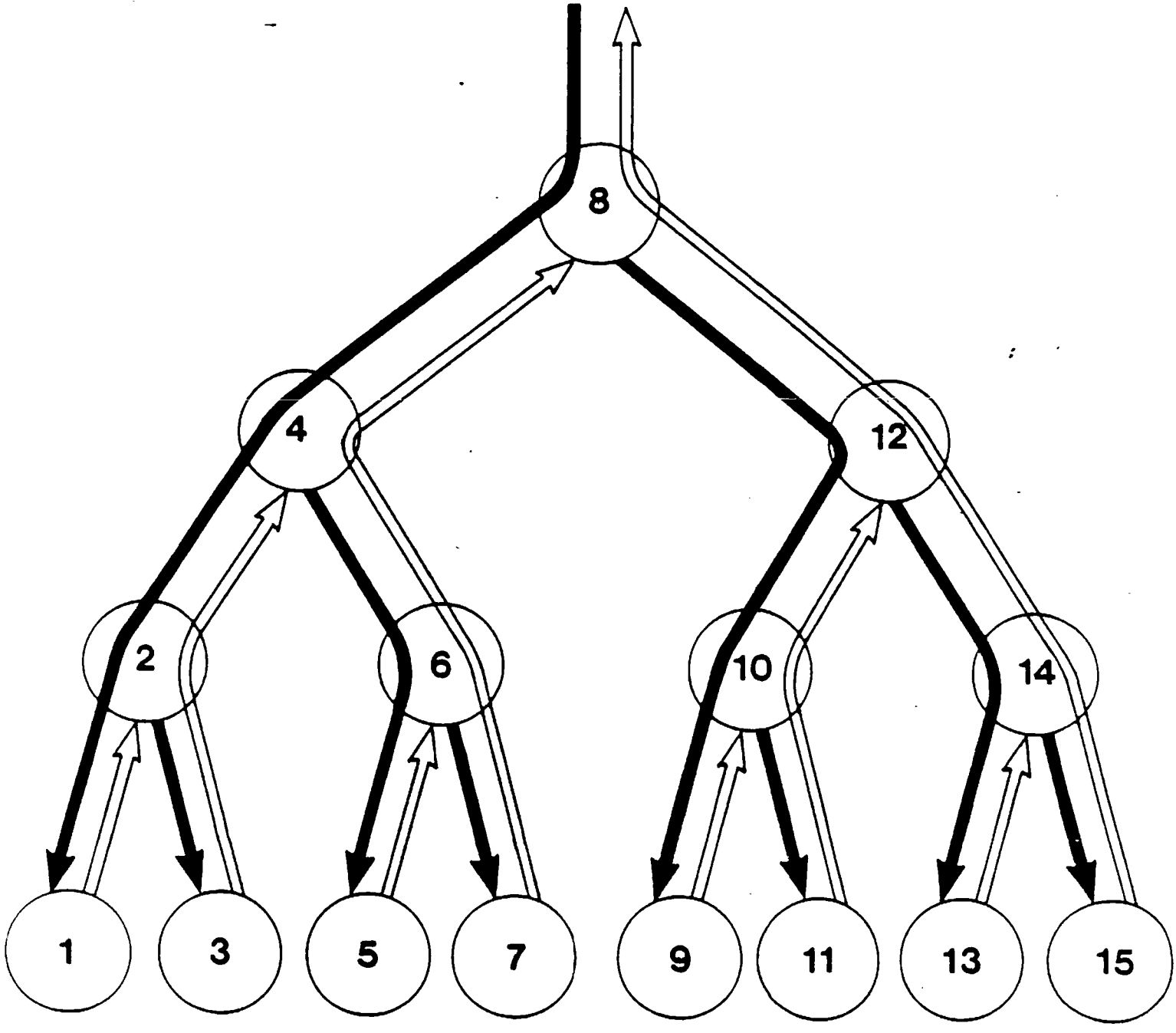

Figure 2: Linear Neighbor Communication - Up and Down Phases

rotation operations facilitate bit-serial arithmetic of arbitrary length, as well as general communication between the 1-bit and 8-bit data paths.

4. Logical Function Group: All sixteen Boolean functions of two bits are provided, with A1 and B1 as operands, and A1 receiving the result.

5. No Operand Group: ENABLE causes all PE's to set EN1. PE's having EN1 clear ignore all other instructions. This allows selective execution in subsets of the PPS. COMPARE generates a comparison of A8 and B8, setting A1 := (A8 = B8), and B1 := (A8 > B8). RESOLVE operates on all enabled PE's having A1 set. It clears A1 in all but the "first" such PE. For NON-VON 1, "first" means first in an inorder traversal of the PPS. This instruction is used to select an individual from a set of PE's, and forms the basis of code to enumerate a set, one element at a time.

6. Communication Group: The BROADCAST1 and BROADCAST8 instructions cause all enabled PE's to receive a bit or byte (respectively) broadcast from the control processor, with the value placed into A1 or A8, respectively. REPORT1 and REPORT8 perform the inverse operation, although the control processor will receive valid data only if precisely one PE has been enabled to execute the report operation. The SEND1, SEND8, RECV1, and RECV8 instructions cause all enabled PE's to send or receive in parallel, each transferring a bit or byte between this PE's A register, and some neighbor's IO register. The three physical neighbors are parent, left child, and right child. Additionally, there are two logical neighbors to each PE, called left neighbor and right neighbor. They are defined to be the PE's visited just prior to and just after this one in a traversal of the PPS. For NON-VON 1, logical neighbors are defined by adjacency in an inorder traversal.

2.2 Secondary Processing Subsystem

The secondary processing subsystem (SPS) of NON-VON is currently conceived of as a collection of perhaps 256 small Winchester disk units, each having a control and connection unit, referred to as an Intelligent Head Unit, or IHU. The IHU provides its disk with comparison logic "at the head", as well as the ability to perform hashing calculations on data being read. Appropriate buffering and error handling are also performed. The SPS provides secondary storage to the PPS, but its primary service is "filtering" data. Selection is performed within the SPS, with only relevant data passed on to the PPS, so that the bandwidth between SPS and PPS is well utilized. The IHU's for an SPS of size 256 would connect to the PPS at all PE's located eight levels below the root of the tree, with the IHU's logically interposed between the I/O switches of the PE's seven and eight levels below the root. Thus each IHU is located above a subtree of the PPS, and these subtrees may all perform disk

I/O simultaneously. This provides an immense bandwidth, allowing efficient use of the massive parallelism available in the PPS.

2.3 Control Processor

The ensemble of PE's cooperates in a single instruction stream, multiple data stream (SIMD) mode [Flynn, 72], under the direction of a general purpose computer attached to the root of the tree of PE's. This control processor (CP) is responsible for the broadcast of instructions through the root and down into the tree, where the instructions are executed in parallel by the PE's. It also handles I/O between the user and NON-VON.

3 Using Abstraction

Experience has shown that in the absence of support from structure and abstraction, one cannot reliably create correct programs for von Neumann machines. We believe that managing the complexity of a parallel processing ensemble is potentially more difficult.

A universal approach to the management of complexity is to raise the level of abstraction and hide low-level details. Programming language designers have developed abstract data types, modules, objects, and message passing. In the area of operating systems, we have monitors and communicating sequential processes. In computer communications, the International Standards Organization has developed the OSI Reference Model for network architectures, defined in terms of seven layers, each providing more sophisticated and abstract services to the layer above it.

In the high level languages under development for NON-VON, we wish to establish a certain degree of hardware independence. In particular, the programmer should not be grappling with issues related to the size of the RAM in each PE. Neither should the programmer explicitly map a record's fields into the PE's RAM address space, nor assign available PE's to hold and process various data. Using information obtained from the programmer's declarations of variables, the runtime system in the control processor will handle such matters.

In the case of multiple instruction stream, multiple data stream processing, the formalisms given for communicating sequential processes seem appropriate. For example, MIMD PE's can be modeled as ADA[2] tasks[Ichbiah et al., 1980] that

---

[2]ADA is a trademark of the US Department of Defense

achieve synchronization during a rendezvous. In the case of SIMD machines such as NON-VON, this formalism does not fit well. Here a PE has just a small amount of data, and no local program. It does not take independent action; it executes the same instructions as all other enabled processors, in lockstep. We have taken a few steps towards understanding the present situation, and have identified three ways of thinking about our data and PE's with a degree of abstraction.

3.1 The Multiple Record Type

The multiple record type is an extension of the formalism given by a Pascal record. It allows reference to an aggregation of data, without giving the superfluous structure implied by an array, file, or list. An instantiation of a multiple record consists of a collection of data, such that each member conforms to the record format given in the declaration. Multiple record declarations serve as explicit indications of data suitable for parallel computation. In the remainder of this paper, the distinction between "multiple record" and "instance of a multiple record" will be suppressed where the context makes the proper meaning clear.

3.2 Virtual PE's

One notion consistent with our desire for independence from the physical details of the PE is that of a virtual PE. If a record requires just a few bytes, we may pack several records into each physical PE. Similarly, if the record is too large to fit in one PE's RAM, it may be split into pieces, and stored in several PE's. In either case, a programmer working in an appropriate high level language sees one record stored per PE. Thus the PE's that are seen by the programmer are virtual, in that they do not have a fixed size determined by the hardware.

3.3 Intelligent Records

Another view of the PPS even further removed from the hardware is that of a collection of "intelligent records". From this perspective, each element in a multiple record is considered to have its own processing power. We may then consider instruction sequences directed from the control processor to the records themselves. For an intuitive example of such a statement sequence, consider

```
All employee records listen to me.
Anyone who has years-employed greater than 20,
   add 1000 to your salary.
```

When expressed in our extended version of Pascal (described later), the previous example would appear as follows:

```
WITH employees DO
  WHERE yearsEmployed > 20 DO
    salary := salary + 1000;
```

Let us examine how such processing might actually be performed by the NON-VON PE's. Since this example is given as an illustration of the functioning of the PPS, the code is written at a higher level than the NON-VON instruction set described earlier. For reasons of perspicuity we have chosen to suppress details such as the implementation of 16-bit addition on a machine having bit-serial arithmetic. We trust the reader is willing to trade rigor for lucidity and conciseness in this instance.

Suppose first that the employee records already have been loaded into the PPS, with one record per PE. When a multiple record is placed into the PPS, a tag field is added to each of the members, so that they may be distinguished from members of other multiple records. For this example, suppose that the tag is stored in the byte of RAM at location 0, the years-employed in RAM location 1, and the salary in the word at locations 2 and 3. Suppose further that the tag designating employee records is "17". We might then have the following assembler-level instructions broadcast down into the PPS by the control processor, for execution in parallel by the PE's.

```
    ; Enable all PE's so they will execute the following instructions.
    ; The hardware effect is to set the enable flag (ENA) in each PE.

ENABLE              ; enable all PE's

    ; Select those PE's which contain employee records:
    ; Load the tag into the accumulator, and compare with 17.  The result
    ; is developed in the EQ and GT flags, which are set to indicate
    ; equality and accumulator greater than operand, respectively.
    ; By storing the EQ flag into the ENA register, those PE's having
    ; tag 17 will remain enabled, while the others will be disabled, and
    ; will not participate in further computation.

LOADB A,0           ; load a byte into the accumulator from RAM 0, the tag
CMP #17             ; compare accumulator with the constant 17
STORE EQ,ENA        ; if equal, remain enabled; otherwise become disabled

    ; PE's still enabled load the years-employed field, and compare with
    ; 20.  Only PE's that contain employee records showing more
    ; than 20 years of employment are permitted to execute further.

LOADB A,1           ; load years-employed
CMP #20             ; compare with the constant 20
STORE GT,ENA        ; if greater, remain enabled
```

```
    ; Finally, for all PE's still enabled, load the salary field (2 bytes)
    ; add 1000, and store the result back

LOAD A,2            ; load bytes 2 and 3 into the accumulator
ADD #1000           ; add 1000
STORE A,2           ; store the result
```

3.4 The Run-time System Layer

The run-time system layer is a collection of functions that provide abstracted data manipulation services to the programmer. These functions generate and issue the low-level NON-VON instructions necessary to accomplish arithmetic, comparisons and matching, I/O, and database operations on multiple records. LISP was chosen as the implementation language for the experimental runtime system. This choice of language was motivated by the freedom from side-effects encouraged by the functional, applicative nature of LISP. This enables relatively simple modifications to be made as our understanding of the problem develops.

The functions that form the run-time system accept arguments which are given in terms of the multiple records. Relational algebraic operations on database tables and views [Date, 1981] are supported, with data referenced by record name and field name. As a brief example, the code to increment (in parallel) the salary of all engineers might be written as:

```
(SETFIELD  (SELECT 'employees 'eq 'jobtitle 'engineer)
           'salary
           (+ (GETFIELD 'employees 'salary)
              1000))
```

Where the functions SETFIELD, SELECT, and GETFIELD are as follows:

(SETFIELD <view> <field> <new value>)
looks up the tag corresponding with the view name, and broadcasts instructions to the PPS in order to enable just those PE's containing data for that view. Then it looks up the absolute PE RAM locations corresponding to the field named, and broadcasts instructions to update the field.

(GETFIELD <view> <field>)
also enables those PE's containing the view, and generates instructions to extract the value of a field from the proper PE RAM locations.

(SELECT <view> <relation> <field> <value>)
enables just a subset of PE's containing a view, based on a comparison (eq, ne, lt, le, gt, ge) of a field and a value.

The function "+" in the context of the example, generates instructions to perform parallel arithmetic in the enabled PE's.

Thus the programmer is insulated from many low-level details. The system finds the records in the PPS, handles the mapping of fields onto PE RAM, and performs low level operations necessary to manipulate information within each PE.

3.5 The Pascal Layer

In an effort to facilitate the use of services provided by the run-time system layer, we have been writing a compiler to translate an extension of Pascal into calls on the run-time LISP functions [Bacon et al., 1982]. The work done so far on extending Pascal must be considered preliminary, as the issues pertaining to the modeling of execution in an SIMD tree machine are not fully understood at this time. Some of the extensions we currently use are described below.

There is one new built-in data type, the multiple record. A variable of type multiple record refers to a collection of records that undergo operations performed in parallel. For example, after the mass administration of a rejuvenating formula, a youth clinic might wish to perform actions suggested by the following code fragments:

```
TYPE
  customer = MULTIPLE RECORD
               name       : PACKED ARRAY[1..30] OF CHAR;
               balance    : REAL;
               ageClaimed : INTEGER;
             END;
   .
   .
VAR
  pigeons : customer;
   .
   .
BEGIN
   .
   .
  { the next statement causes parallel operations }
  WITH pigeons DO
    BEGIN  balance    := balance - 400.00;
           ageClaimed := ageClaimed - 10
    END;
```

The five unary operators that perform vector merge operations on Boolean values developed in the intelligent records are ANY, ALL, NONE, SEVERAL, and ONE. They may be used to control the selection of alternatives in an IF statement. ANY is true if the condition holds in one or more member of the multiple record, ONE is true if the condition holds in precisely one member, while SEVERAL means more than one. To continue the previous example, we could write:

```
IF ANY (pigeons.ageClaimed > 40)
   THEN Writeln('There is still money to be made.')
```

Two new statements have been added. The first has the form:

```
WHERE <Boolean expression> DO <statement>
  ELSEWHERE <statement>
```

(The ELSEWHERE portion is optional.)

The effect of this statement is to calculate the Boolean expression in all currently enabled intelligent records. Those that satisfy the Boolean execute the statement after DO, while those enabled records which do not satisfy the Boolean execute the statement after ELSEWHERE.

The FOR EACH statement is used to iterate sequentially through the members of a multiple record. It has the form:

```
FOR EACH <multiple-record identifier>
  DO <statement>
```

Continuing the rejuvenation example, we may have the following code:

```
Writeln(' Customers who have been reduced to childhood:");

FOR EACH pigeon DO
  IF pigeon.ageClaimed < 18 THEN Writeln(pigeon.name)
```

In addition, other statements have been extended semantically to apply to multiple records. For the CASE statement, if its controlling expression refers to a multiple record, all the members will be processed, each according to the case it satisfies. This use of CASE is equivalent to a linear nesting of WHERE ... ELSEWHERE statements. For the WHILE ... DO and REPEAT ... UNTIL statements, if the controlling Boolean refers to a multiple record, the body of the loop will be executed in parallel, in all enabled members of the multiple record.

4 Storage of Records Larger Than a PE's RAM

Let us reconsider the example given earlier,

```
WITH employees DO
  WHERE yearsEmployed > 20 DO
    salary := salary + 1000
```

The assembler-level code previously given for this example was based on the assumption that an employee record fits into a physical PE. If this is not true, the record must be subdivided, with portions stored in several PE's. Records so stored are termed spanned records. Records that are spanned across physical PE's introduce complications into the processing. For instance, suppose that in the present example, the years-employed and salary fields are not in the same PE. After selecting the PE's having years-employed greater than 20, how may we find the corresponding salary fields? As another example, suppose we wish to compare the contents of one field against another in each member of some multiple record. If the two fields from each record are not in the same physical PE, communication between pairs of PE's will be required. How may this be done in parallel, using a single instruction stream?

4.1 Linear-neighbor Spanned Records

One good choice of allocation patterns is to place subparts of spanned records in PE's which are adjacent by their linear-neighbor connections. In the simpler non-spanned case, we addressed fields in PE RAM by their byte offset. Now we use an offset counted in PE's from the head of the record, and then the

byte offset within that PE. If the tag is placed in RAM cell 0 in the head of each record, RAM cell 0 in the other PE's must be cleared to 0 so that record heads may be found unambiguously. Suppose for the employee example, that each record spans three PE's. Suppose further, that the years-employed field is in the first PE at offset 1, and the salary field is in the third PE at offset 4. Then the previous example would appear as follows:

```
    ; Enable all PE's so they will execute the following instructions, and
    ; clear the I/O register, as it will be needed for linear-neighbor
    ; communication later.

ENABLE              ; enable all PE's
LOAD IO,#0          ; clear the I/O register


    ; Select those PE's which contain heads of employee records.

LOADB A,0           ; load a byte into the accumulator from RAM 0, the tag
CMP #17             ; compare accumulator with the constant 17
STORE EQ,ENA        ; if equal, remain enabled; otherwise become disabled


    ; Now select for years-employed greater than 20

LOADB A,1           ; load years-employed
CMP #20             ; compare with the constant 20
STORE GT,ENA        ; if greater, remain enabled


    ; For all records whose heads are still enabled, we wish to move over
    ; two PE's to the right.  First we will put a 1 into the I/O register
    ; of PE's which are enabled (recall all I/O registers were cleared at
    ; the beginning).  Then we will send this "mark" to the right twice,
    ; and finally use it to set the enable register appropriately.

LOAD IO,#1          ; mark the I/O register of all PE's currently enabled
ENABLE              ; everyone listen
SEND RN             ; move the mark to the right neighbor
SEND RN             ; move the mark to that PE's right neighbor
STORE IO,ENA        ; those PE's which are now marked stay enabled.


    ; Finally, for all PE's currently enabled, i.e. those containing the
    ; third portion of records for employees employed more than 20 years,
    ; load the salary field (2 bytes), add 1000, and store the result back

LOAD A,4            ; load bytes 4 and 5 into the accumulator
ADD #1000           ; add 1000
STORE A,4           ; store the result
```

Comparison of fields from different PE's of a spanned record may be handled in a similar fashion. For example, to compare the salary field with some field in the head of employee records, one might proceed in this fashion:

```
        ; The salary field is in bytes 4 and 5 of RAM in the 3rd PE's.
        ; We will take bytes 4 and 5 from  A L L  PE's, and send them left
        ; two times.  The result will be that in PE's that have heads of
        ; employee records, the I/O register will contain the corresponding
        ; salary value.

    ENABLE              ; enable all PE's
    LOAD IO,4           ; load bytes 4 and 5 into the I/O register
    SEND LN             ; send to left neighbor
    SEND LN             ; send to that PE's left neighbor

        ; Now select those PE's which contain heads of employee records.

    LOADB A,0           ; load a byte into the accumulator from RAM 0, the tag
    CMP #17             ; compare accumulator with the constant 17
    STORE EQ,ENA        ; if equal, remain enabled; otherwise become disabled

        ; At this point, only heads of employee records are enabled, and in
        ; those PE's, the I/O register contains the salary from that record.
```

The preceding examples have all assumed fixed length fields to simplify the code presented. In fact, NON-VON can work with varying length fields also, assuming a table of offsets to fields is stored in the head of each record. Storing these offsets in each PE gives the one level of indirection needed to process variable length fields on an SIMD machine.

4.2 Tree-shaped Spanned Records

The linear-spanned record storage scheme described above has a theoretical disadvantage in that communication between fields of a record requires time that is linear in the length of the record.[3] One possible improvement would be

---

[3]Note that we are considering the time required for a linear neighbor communication to be constant, since it is a primitive operation provided by our hardware within the usual instruction execution time. In another model, one could assign a time complexity O(log |PPS|) to this communication, where |PPS| is the number of PE's in the Primary Processing Subsystem.

to allocate spanned records in subtrees, as illustrated below. Unfortunately, this would only use the levels of the PPS near the leaves, while not using those PE's in the top portion of the tree. Communication within a record in this case would occur in time proportional to the logarithm of the record length. It should be noted that the constants are such that in practical situations, operations within subtree-spanned records might not be faster than those in linear-spanned records.

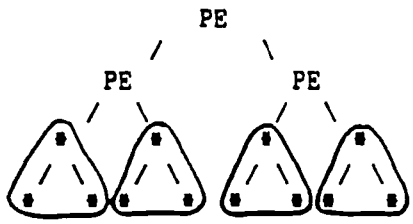

Figure 3: Subtree allocation.

Some problems are solved particularly efficiently in an architecture providing tree-structured communication. A disadvantage shared by linear- and subtree-spanned records described previously, is that a tree structure of communication between records is not available. This capability may be obtained, however, through a generalization of the subtree-spanned record. One generalization has the same tree-shaped allocation pattern for each record, but records are located at arbitrary positions in the PPS, not only at the leaves. For example, two records which each span three PE's might be allocated as depicted below:

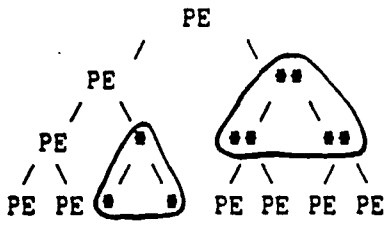

Figure 4: Two bushes

We have termed tree-shaped structures such as these bushes. A k-bush is a collection of k PE's forming a complete, full binary tree. The number of PE's required to form a bush is one less than a power of two, which suggests that bushes are less space efficient than linear-spanned records. This relative disadvantage arises from a phenomenon very similar to "internal fragmentation" in a paging environment. In the case of linear-spanned records, we only expect half of the space in the last physical PE to be wasted. In bushes, however, the waste is asymptotically greater. Maximal waste occurs when the space required by a virtual PE is one byte more than will fit in a full bush, in which case the next larger size must be used. This larger bush will

consist of one more than twice as many PE's, and so will be slightly more than 1/2 empty. On the average, then, one quarter of the space in a bush will be wasted. For linear-spanned records, the wasted space will be independent of the record size, but in the bush pattern of allocation, this is not so. Equivalently, we may say that the asymptotic average space wasted is zero for linear-spanned records, but is about 25% for bush-spanned records.

Bushes may be organized in two ways. Unkempt bushes are scattered throughout the PPS in arbitrary positions, as in the previous figure. For algorithms requiring tree-structured communication between records, landscaped bushes are desired.

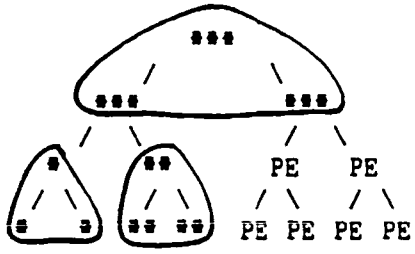

Figure 5: Three landscaped bushes

A landscape is a collection of k-bushes that forms a full (k+1)-ary tree. Landscaped bushes provide tree structured communication among records. An algorithm for calculating all partial sums will be presented as an example of the use of this communication to obtain asymptotically faster time complexity (excluding I/O) than available on sequential machines.

4.3 A Speedup From Tree-structured Communication

Any algorithm for generating n partial sums on a sequential machine must form each of these n sums. Even though the work charged to each sum can be done in constant time [Browning, 1980], there are n sums; the total time required is O(n). In NON-VON, as in the tree machine developed by Browning, the formation of partial sums can be overlapped, so that the aggregate time required for the calculation is O(log n). Browning's tree machine has a multiple-instruction stream, multiple data-stream (MIMD) organization, as well as message buffering between PE's. We present a derivative of her algorithm for NON-VON, which operates with the same asymptotic time complexity, which is SIMD, and which does not need message buffers. Starting with data arranged according to an inorder traversal of the PPS, the algorithm develops all partial sums so that the k^{th} PE receives the sum of values 1 through k.

The computation occurs in two phases. During the first phase intermediate sums propagate up the PPS tree from leaves to root. In the second phase, sums move back down the tree. For this discussion, we refer to the levels of the

PPS by number, with 1 at the root and k at the leaves.

In the up phase, summing occurs level-by-level, starting with the leaves. Each PE on the currently selected level receives the value held by its left child and adds it into its own value. Then it receives the value held by its right child, and places the sum of that and its own value into a variable available for its parent to read from the next level.

In the down phase, summing also proceeds level-by-level, starting with the root. PE's on the selected level take the value received from their parent, and pass that value on to their left child. Then they add it into their own partial sum, and also pass this new sum on to their right child.

For expressing the algorithm in the extension of Pascal, each intelligent record contains two variables, named data.self and data.io . The algorithm begins with the variable self in each intelligent record containing that record's contribution to the sum. The result of the computation is developed in that variable, overwriting the initial value. Three built-in functions are assumed in the extended Pascal. EnableLevel(i) enables those intelligent records in level i of the landscape of records. Receive(myVar, nbr, itsVar) operates in parallel, in all enabled records. It moves the contents of the neighbor's variable itsVar into the local variable myVar, where permitted neighbors are identical to those for the NON-VON instruction set described earlier. Send(myVar, nbr, itsVar) allows all enabled records in parallel to send a copy of myVar's contents to itsVar in the specified neighbor. The implementation of these functions using the communication primitives provided by the PPS hardware is straightforward, and as used in the following code, each is a constant time operation.[4]

[4]In general, EnableLevel(.) is a log time operation.

```
WITH data DO
 BEGIN

 { initialization }
 io := self;

 { up phase }
   FOR i := k-1 DOWNTO 1
     BEGIN
       EnableLevel(i);
       Receive(io, leftChild, io);
       self := self + io;
       Receive(io, rightChild, io);
       io := self + io
     END;

 { down phase }
   EnableLevel(1);
   io := 0;
   FOR i := 1 TO k-1
     BEGIN
       EnableLevel(i);
       Send(io, leftChild, io);
       self := self + io;
       Send(self, rightChild, io)
     END;
   EnableLevel(k);
   self := self + io

 END { WITH data }
```

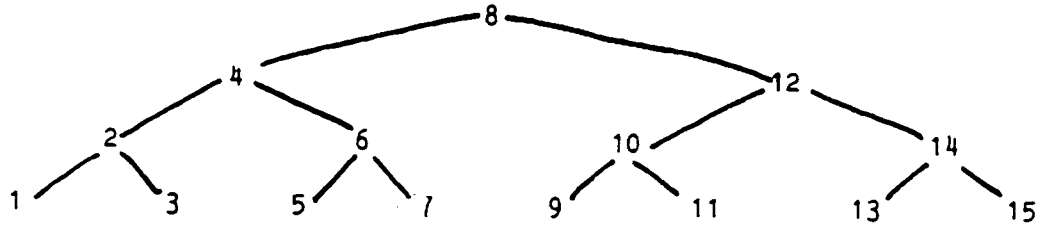

Figure 6: All Partial Sums -- Original Data

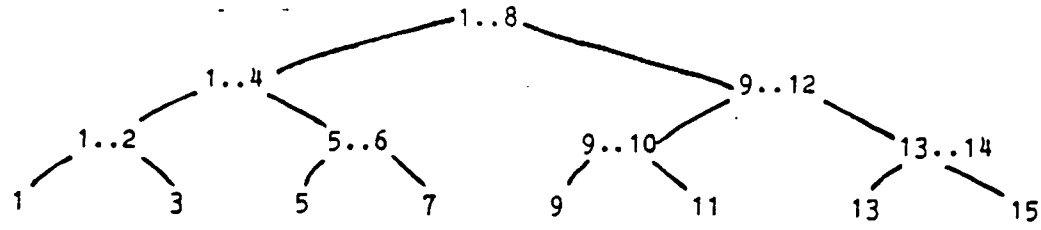

Figure 7: All Partial Sums -- After Up Phase

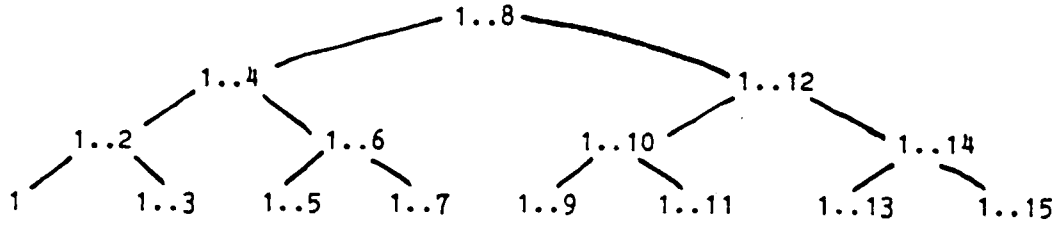

Figure 8: All Partial Sums -- After Down Phase

4.4 Bushes Are Not Binary

The algorithm given for computing all partial sums is designed specifically for a binary tree. Thus it works for records stored in 1-bushes: bushes consisting of one physical PE. However, a bush consisting of more than one PE has more than two children. For example, a 3-bush has a root and two leaves. From these two leaves, the 3-bush obtains four children. A 7-bush has eight children. Thus it is desirable that programs for NON-VON not be written specifically for binary trees. Fortunately, we have found it usual for 1-bush algorithms to have a natural generalization to algorithms for the k-bush case. As an example, we note that the All Partial Sums algorithm given previously works for k-bushes, provided that certain conditions are met. First, the data being summed must be located in the root of each bush during certain times. If the field containing the value being summed happens to be in some other PE, it must be moved up to the root of the bush during (or prior to) the calculation. Second, PE's other than roots of bushes must have space set aside to hold a temporary value that is initialized to 0. Third, the data are placed into bushes in an appropriate generalization of the inorder pattern. The generating rule for "extended inorder" on a 2k-ary tree is "visit the left-most k children from left to right, then visit the self, then visit the right-most k children from left to right".

Given these three conditions, the previous All Partial Sums algorithm works for bushes consisting of 2k-1 physical PE's. The sums present in a landscape of 3-bushes after the up-phase and down-phase of the algorithm are illustrated

in the next two figures.

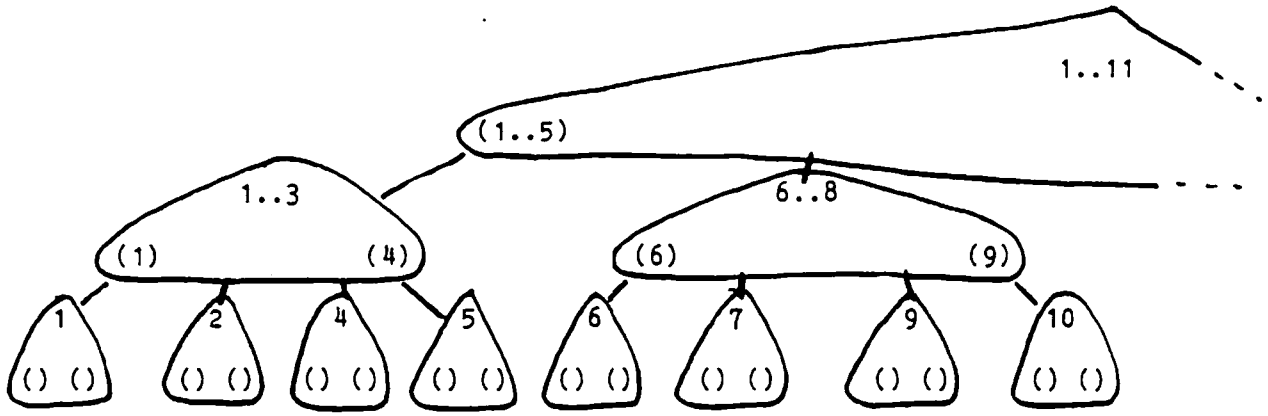

Figure 9: 3-Bush Partial Sums -- After Up Phase

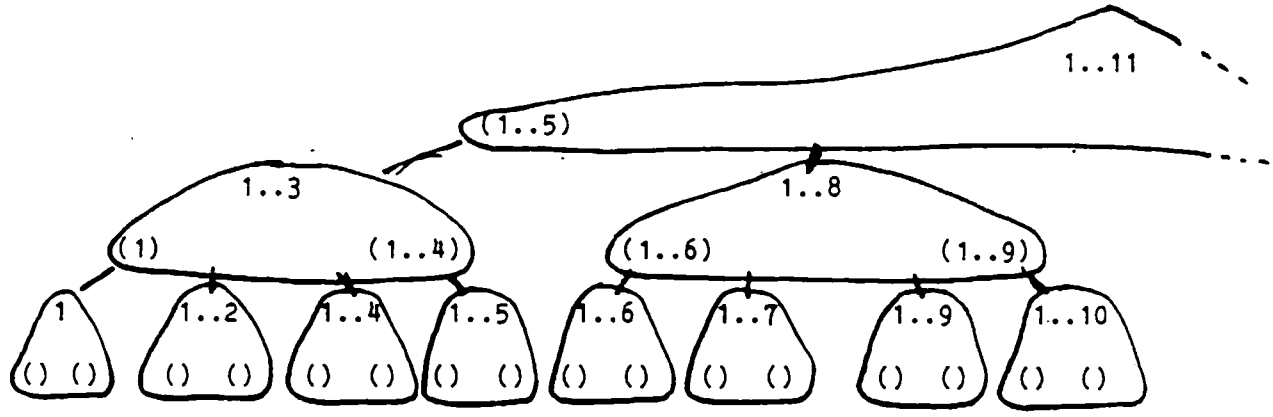

Figure 10: 3-Bush Partial Sums -- After Down Phase

4.5 Allocation of PE's To Landscapes

During normal operation, it is expected that more than one multiple record will reside in the PPS at any given time. Thus it will be necessary to find free space to hold a multiple record that is to be loaded into the PPS. The following algorithm will determine whether there is enough space of the proper shape to hold a landscape of a given size, and if so, will identify the root of that landscape. The running time of the algorithm is no worse than $O(\log^2 n)$, where n is the number of PE's in the PPS.

In the Pascal extension, the built-in function Resolve takes as argument a variable in a multiple record. It uses the hardware RESOLVE instruction to identify just one member for which the variable is non-zero. The presence of

responders, indicated by the condition of a control wire in the control processor, is returned as a Boolean value. Procedure MarkEmpty puts a temporary record in registers of all PE's. The record contains two Boolean variables, self and io, with self initialized to TRUE in PE's that are empty.

1. Let the levels in the PPS be denoted 1 through L, with 1 for the root and L for the leaves. Let k be the number of levels in the PPS needed to hold the landscape.

```
startLevel := L;
responder := FALSE;
WHILE (NOT responder) AND (startLevel >= k) DO
 BEGIN
   MarkEmpty;

   {AND the mark up k-1 levels:}
   FOR level:=1 TO k-1 DO
    BEGIN
      EnableLevel(startLevel - level);
      Receive(io, leftChild, self);
      self := self AND io.
      Receive(io, rightChild, self);
      self := self AND io
    END;

   {Was an entire landscape empty?}
   responder := RESOLVE(self);

   { If there is a responder, stop.  It is the root of an available
     landscape of the requisite size.  Otherwise loop back to look
     for a landscape starting one level higher in the PPS. }
   IF NOT responder
     THEN startLevel := startLevel - 1;
 END { WHILE }

 { If responder is true, the search succeeded.  Otherwise, there is
   not sufficient space for the landscape. }
```

2. After a landscape's root has been identified, the heads of bushes in the landscape may be marked in O(k) time by proceeding downward from the root of the landscape, marking all PE's in the appropriate levels.

The same algorithm may be used for the allocation of random bushes. In this case, the landscape-size parameter is set to the desired size of one bush, and the procedure is used repeatedly until as many bushes as needed are identified and marked as not-empty.

It may be noted that the allocation of linear-spanned records may be performed more rapidly. The time required is O(r), where r is the number of physical

PE's required to hold one record. A method that achieves this time complexity is to mark all available PE's, then propagate that mark through left neighbor connections r-1 times, destroying the mark in transit if it passes through a PE which is not available. After r-1 of these "left shifts", the marks will be present in all PE's that can serve as heads of the linear-spanned records. The potential records so identified may overlap, but this need not cause difficulties. Linear-spanned records do not use tree-neighbor communication. They are loaded with data one at a time.[5] As each PE is filled, any mark indicating "potential head of record" can be cleared, so that the next available empty record may be located in constant time, using the RESOLVE instruction. One other detail is that it is necessary to prevent a record from spanning across subtrees, since the subtrees are loaded in parallel.

5 Storage of Records Smaller Than a PE's RAM

Despite the great bandwidth between NON-VON's Primary and Secondary processing subsystems, internal operations tend to be faster than I/O. To solve problems having more data than the number of PE's in the PPS, it may be necessary to take several passes over the data. External sorting is a well known example of an analogous situation in the sequential machine environment. In NON-VON it is natural to place one record per PE, but if each record is significantly smaller than the PE's RAM, the number of external processing passes through the data can be reduced by storing several records per physical PE. This will yield a decrease in the I/O time, although the internal processing time may increase by a factor proportional to the number of records stored per physical PE; the "packing ratio".

In processing a record, certain state information is developed in the registers of the physical PE which contains it. If several records are to be stored in a PE, it may be necessary to provide a small amount of RAM with each record to save the register contents. Thus at any instant, one record is "exposed", in that its state is in the PE's physical registers, while the other records stored in that PE are "buried", with their state information hidden in the PE's RAM. The collection of all currently exposed members of a multiple record is called a "slice". In algorithms requiring the interaction of each record with all others, one may expect a time penalty proportional to the packing ratio, because if k records are packed per physical PE, it may be necessary to repeat every operation k times, once for each slice. An illustration of this is given by deletion of duplicates, from the area of database manipulation. We choose one record, report it out of the PPS, and broadcast it back in, matching for duplicates, and destroying any which are found (including the original just read out). This process is repeated until there are no more records to be reported out. Suppose that there are n

---

[5]The intelligent head units may load their subtrees of the PPS in parallel, but under each IHU the loading is sequential.

distinct records. Each record that is not a duplicate is read out to the CP. This is O(n). For each such record, it must be matched against all p slices, which is O(p). Thus the run-time is O(pn), which shows that in this example the cost of packing is a multiplicative factor equal to the packing ratio.

6 Another Associative Storage Allocation Pattern

Previous examples have illustrated the use of linear-spanned records and landscaped bushes. One other common storage organization found in NON-VON algorithms uses the PPS as an associative store. In this case, bushes need not be landscaped; random bushes serve just as well. The increased flexibility of this allocation pattern may permit more efficient use of the space available in the PPS.

The transitive closure G^* of a digraph G=(V,E) has an edge x-->y iff there is a directed path from x to y in G. We will let n denote the number of vertices |V| in G. The number of edges |E| in G falls in the closed interval $[0,n^2]$, as each vertex may be the initial endpoint of edges to each other vertex, as well as of a self-loop. Similarly, n^2 bounds the number of edges in G^*.

There are several transitive closure algorithms for sequential machines mentioned in [Browning, 1980]. Perhaps the most widely known is Warshall's Algorithm, which uses an adjacency matrix for input and output, and has time complexity $O(n^3)$. The best time complexity mentioned by Browning for sequential machines is $O(T \log^2 n)$, where T is the time required to multiply two matrices. Currently T is approximately $O(n^{2.5})$, so the best sequential algorithm complexity is about $O(n^{2.5} \log^2 n)$. For her MIMD tree machine, Browning achieves a run time of $O(n^2)$, using $O(n^2)$ processors.

As may be expected, NON-VON can calculate the transitive closure of a graph with better asymptotic time complexity than von Neumann machines, meeting the trivial $O(n^2)$ lower bound imposed by I/O requirements in a tree machine.

The first NON-VON algorithm given for this problem is a modification of the Floyd-Warshall algorithm. It achieves time complexity $O(n^2)$, using $O(n^2)$ virtual PE's. This provides an example supporting an observation we have made for certain algorithms: the highly area-efficient simple processors and communication hardware used in the NON-VON PPS can equal the asymptotic speed of an MIMD tree machine with powerful PE's and communication primitives.

The second NON-VON algorithm has $O(n^2 \log n)$ worst-case time complexity. It uses random bushes, and a technique not unlike Browning's. It is quite space efficient, however, requiring n bushes, each containing just n bits of RAM. As an aside, it may be noted that this algorithm will also work using n linear-spanned records, but with a time degradation to $O(n^3)$. The space required remains $O(n^2)$, but with better constants than for the bush implementation because of the "internal fragmentation" effect mentioned

earlier.

6.1 Transitive Closure 1

The first transitive closure algorithm for NON-VON is a parallelization of the Floyd-Warshall algorithm. Each PE represents an edge in the complete graph K_n. Specifically, it holds integers a pair of integers representing the initial and terminal endpoints of the edge, as well as a Boolean value indicating whether the edge is a member of the closure. The sequential version of Floyd-Warshall sets the Boolean in the following way:

```
FOR k:=1 TO n DO
  FOR i:=1 TO n DO
    FOR j:=1 TO n DO
      exists(i,j) := exists(i,j) OR ( exists(i,k) AND exists(k,j) )
```

The parallel version used by NON-VON uses n^2 virtual PE's, each containing one of the exists(*.*) Booleans. In each of n iterations, for some fixed k, all of the Booleans given by exists(*,k) and exists(k,*) are broadcast into the PPS, in O(n) time. All n^2 PE's are working: PE(i,j) listens for exactly two Booleans, namely exists(i,k) and exists(k,j). If they are both true, then PE(i,j) sets exists(i,j) true, and this happens simultaneously for all i and j.

```
FOR k:=1 TO n DO
  BEGIN
    FOR i:=1 TO n DO
      BEGIN  read exists(i,k);
             broadcast exists(i,k)
      END;
    FOR J:=1 TO n DO
      BEGIN  read exists(k,j);
             broadcast exists(k,j)
      END;
    exists(i,j) := exists(i,j) OR ( exists(i,k) AND exists(k,j) )
  END
```

In the sequential version, each iteration of the loop for k has time complexity $O(n^2)$, limited by the time for the nested loops for i and j, so the algorithm is $O(n^3)$. In the parallel version for NON-VON, each iteration of the loop for k has time complexity O(n), which is limited by the I/O time. The exists Booleans are updated in constant time, because the PE's perform this in parallel, n^2 at once. As an aside, we note that similar algorithms will multiply matrices and solve problems such as all pairs shortest path, all in $O(n^2)$ time. Sequential Floyd-Warshall has been classified in [Aho, Hopcroft, Ullman, 1974].

6.2 Transitive Closure 2

For the second algorithm, n virtual PE's are used, each having (at least) n bits of RAM. The virtual PE's are numbered with the vertices of G. If bit y is set in virtual PE x, that represents an edge x-->y in G. The virtual PE's may take the form of bushes or linear-spanned records, but a difference will be seen in the algorithm's time complexity, which will be $O(n^2 \log n)$ for the bush variant, and $O(n^3)$ for the linear-spanned version, reflecting the differing cost of communication within each of these types of spanned records.

Step 1. Initialize the PE's.

First we place into each PE its positional number in an inorder traversal. This can be done hierarchically on the n bushes in O(log n) time, as follows. Let the number n of virtual PE's be 2k-1. The numbering is performed level-by-level in the PPS. The root is labeled with k. It labels its left child as k - k/2, and its right child as k + k/2. They label their left children with their own value minus k/4, and their right children as their value plus k/4. The next level of labeling is performed by subtracting and adding k/8. The process continues until the leaves of the PPS are labeled. Next the RAM cells are cleared. Since there are n bits of RAM in each virtual PE, this step can be done in O(n) time.

Step 2. Broadcast the edges of G.

For each edge x-->y in G, select the PE with virtual number x, and set its RAM cell y. The time complexity of this step is bounded by the number of edges in G, which is at most $O(n^2)$.

Step 3. Develop the transitive closure.

First push all the edges of G onto a stack.

WHILE stack not empty DO

1. Pop stack top into xy, send xy to output; it is part of the solution.
2. Select PE x; set its RAM cell y. { so that x-->y will not be generated again }
3. Select all PE's except x. In parallel, mark all those for which RAM cell x is set and y is clear, then set their RAM cell y.
4. Associatively enumerate the marked PE's, reporting out their inorder numbers k, and push each new edge k-->y onto the stack.

Step 3.3 can be understood from the point of view of PE k as "I know that I can get from myself to x; I have just heard that we can get from x to y,

therefore we can get from myself, k, to y." The only edges that result to be read out in step 3.4, are those of the form k-->y. They were not previously generated, because the test of 3.3 checked that RAM cell y was clear. Edge k-->y can only be created by virtual PE k. Thus edges are not generated more than once.

Each iteration of this loop uses one edge from the stack. Every edge on the stack is an edge in the transitive closure. There are no duplicates produced. Thus it might appear that the time complexity of this step is linear in the number of edges of G^*, $O(n^2)$ in the worst case, but this is not so. In order to do the comparisons in part 3.3, it is necessary to bring the contents of RAM cells x and y into the physical PE that heads each virtual PE. For bushes, this can be done in O(log n) time. For linear-spanned virtual PE's, it takes O(n) time. Thus in the case of bushes, the algorithm has time complexity $O(n^2 \log n)$, whereas for the linear-spanned case the algorithm complexity is $O(n^3)$.

7 Conclusion

This paper has discussed programming language constructs, data organizations, and algorithms for the NON-VON Primary Processing Subsystem. We observed that the conception of problems in terms of virtual PE's and intelligent records tends to make certain programming tasks easier on NON-VON than on conventional sequential processors. For example, direction of instruction sequences to intelligent data records obviates the need for generation and manipulation of data structures, removing an entire area of program design activity. The runtime-layer of functions performing low-level services insulates the programmer from details of the machine's architecture. The Pascal extensions provide a convenient and natural means for expressing parallel activities: substantial speedup is achieved without the necessity of extracting implicit parallelism, yet awkward low-level specification of parallel activities is avoided. Aspects of SIMD programming common to associative architectures in general are added as new statements and extensions to fundamental semantics, while instructions that depend on the particular architectural features of NON-VON are encapsulated in built-in functions. As with any conventional compiler, the programmer is protected from details such as the explicit mapping of variables into memory and registers. Low-level issues which arise only in a parallel environment, such as limited RAM in each PE, are also handled invisibly.[6] Because of the fineness of NON-VON's granularity, this is not excessively wasteful of computing resources. For contrast, one may consider the devastation that would result from the wastage of 25% of the PE's

---

[6]Programs for more than a PPS-full of data still require explicit handling by the programmer, analogous to "external" algorithms for conventional machines. The role of NON-VON's Secondary Processing Subsystem for this domain of programming is not discussed in the present paper.

in ILLIAC IV.

Two principal sources of speedup gained by NON-VON have been noted. The first is derived from the associative processing capability, which allows the inner loop of many sequential algorithms to be performed in constant time, rather than linear time. The second is a result of tree-neighbor communication patterns, by which mathematically associative vector merge functions can be computed in O(log n) rather than O(n) time. We observe that compute-bound algorithms for sequential machines can achieve I/O-limited computation rates on NON-VON, suggesting that wires are the fundamental limitation of our architecture.

8 Bibliography

Aho, Hopcroft, Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.

John Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs", Communications of the ACM, Vol. 21, No. 8, August 1978, pp. 613-641.

David F. Bacon, Monnett Hanvey, Reynaldo W. Newman, Allesandro Piol, NON-VON Pascal, Columbia Computer Science Technical Report, December 12, 1982.

E. F. Codd, "A relational model of data for large shared data banks", Communications of the ACM, Vol. 13, No. 6, June 1970, pp. 377-387.

C. J. Date, An Introduction to Database Systems, Addison-Wesley, 1981.

Michael J. Flynn, "Some Computer Organizations and Their Effectiveness", IEEE Transactions on Computers, Vol. c-21, No. 9, September 1972, pp. 948-960.

Jean Ichbiah et al., Reference Manual for the ADA Programming Language, Proposed Standard dOcument, US Department of Defense, July 1980.