AUTOMATIC PARALLELIZATION OF SEQUENTIAL CODE: A SIMPLIFIED MODEL

David M. Prepejchal

CSCI 499H

April, 1986

The recent emphasis on pushing forth in an attempt to create a fifth-generation supercomputer has put most of the effort into designing the hardware required by such a machine. The speeds of such devices, which are largely parallel in their archetectures, are phenomenal. In comparison, however, there have been few widely accepted successes in the development of a software structure to utilize the power these supercomputers offer.

At present, a variety of successful efforts in parallelization of sequential computer code are noted. However, few of them aim towards the fully automatic execution of this task. Largely, they are approached with the premise of putting some of the burden onto the programmer. In such semi-automatic systems, the programmer is expected to offer a considerable indicaton to the machine of how to go about partitioning the code into concurrently executable blocks. It is widely felt that this awareness on the programmer's part is essential to maximizing the efficiency of the resources available. It is also felt that in the future, efficient, fully automatic parallelization will make the need for such compromises unnecessary. Until languages and practices designed to exploit a problem's potential for parallelization are feasible, the bulk of the responsibility for maximizing processing lies with the machine, on both hardware and software levels.

The purpose of this paper is not to produce a commercial alternative to the problem, but to outline some of the obstacles encountered and overcome in an attempt to produce a simple model of a software structure designed to automatically parallelize serial computer codes in a realistic machine. It is based on small-scale precepts, and in no way was intended to suggest a commercially feasible model. It does, however, introduce solutions to various problems universally encountered in designing a

tightly-coupled multiprocessor system.

The assumptions made at the beginning of the project were modest and fall within the range of present computing technology. Certain assumptions dealing with hardware and low-level system capabilities were assumed feasible irregardless of their presence or absence in actual machines. These assumptons have provided a physical basis for this model, but do not apply strictly to it.

INITIAL ASSUMPTIONS CONCERNING THE HARDWARE SUBSTRUCTURE

The only assumptions concerning a hardware structure are that the system requires a shared central memory, that there exists a single control processor, and that there is at least one "other" processor. The single shared main memory has distinct advantages over local memories for each processor. There is a marked cutdown on cross checking for current values of variables which are being accesses/updated by several processors. Now, under most circumstances, only one instance of a given variable will exist in memory at any one time. A trade-off involved with this is that processors requesting exclusive control over a variable still need to check if it is available and in the desired state at the time it is requested, however this situation is inherent in the problem of parallel computation, and will be accepted. The model to be described has been worked away from the need for local memories for each processor, and this seems to increase the overall efficiency of the system.

The single control processor which governs the work of the several "slave" processors is assigned the duties of the initial processing of the stream of sequential code. As it passes over the code, it sets up the necessary control structures and partitions the code in such a way as to make it concurrently processable. In this model, this first pass is done sequentially, yet it is a logical extension to assume that by carrying this process out in parallel, a potential severe bottleneck in the system's efficiency could be avoided.

Since the control processor partitions the code as it generates pseudo-object codes, it seems that all the information necessary for execution is included in the object forms of the instructions. Therefore, no assumptions

concerning direct inter-processor communications will be made at the hardware level. Ideally, the need for such communications could be avoided by an essentially data-driven system, where all information required at any time is available in the shared memory.

THE SIMPLE WORKING LANGUAGE TO BE USED IN THE MODEL

In its ultimate form, the system indicated by this model would be language independent, or at least modifiable towards any common application language desired. However, in an attempt to avoid the intricacies of handling any signifigant language in favor of demonstrating the methods of data manipulation in this parallel environment, a simplified, explicit language has been created. It is simplified in the respect that it has basic capabilities, but would be a real chore to do any serious programming in. It is explicit in that most instructions allow for distinct "source" operands and a distinct "destination" operand. The general instruction set of this language follows:

```
ADD    op1,op2,op3          op1 = op2 + op3
SUB    op1,op2,op3          op1 = op2 - op3
MULT   op1,op2,op3          op1 = op2 * op3
DIV    op1,op2,op3          op1 = op2 / op3
LOAD   op1,op2              op1 = op2

DO     k,op2,op3            DO WHILE;  k = op2 TO op3
ENDDO                       ENDDO

COMP   op1,op2              COMPARE op1 AND op2
BRNCH op1,CC                BRANCH TO op1 IF CC SATISFIED
```

The above language is designed to be barely sufficient for numerical

processing. The basic instruction format is indicated by the first group, where op1 represents the destination of the result of f(op2,op3). The next group is a specialized pair of looping instructions whose special handling will become evident later. The third group is based on the assumption that condition codes will be set and accessible based on comparisons.

In subsequent examples, there is also a great deal of ambiguity with respect to the actual form of operands. As a generalization, they may referr to high-speed registers, actual storage locations, or array elements by ordered subscripts with equal efectiveness. This discrepancy will not cause any great problems if all resources are considered as shared among the processors. A safe way to think of an operand is as a refference to some location within the central shared memory, avoiding details which tend to rely on a more detailed hardware structure. In cases where operands appear as constant "literal" values, these should be taken at face value. Array subscripts may be taken as either constants or as operands of previous instructions.

CONCERNING STATIC AND DYNAMIC VARIABLE CHARACTERISTICS

The organization of the general class of instructions lends itself to a natural distinction between variable types. Variables, which appear in instructions in the form of operands, may be considered static or dynamic in nature. Static variables are those which may be taken at face value by any processor interested. Static variables have at no time previous to the current instruction had their values modified in any way. In this category are those "literal constant" values, and as indicated by the instruction conventions, variables which have appeared as operands only on the right side of an instruction. Hence, a variable showing up as op1 will not be static. Static variables are of a "read only" type which are safe for use as op2 or op3 in an instruction by any processor at any time.

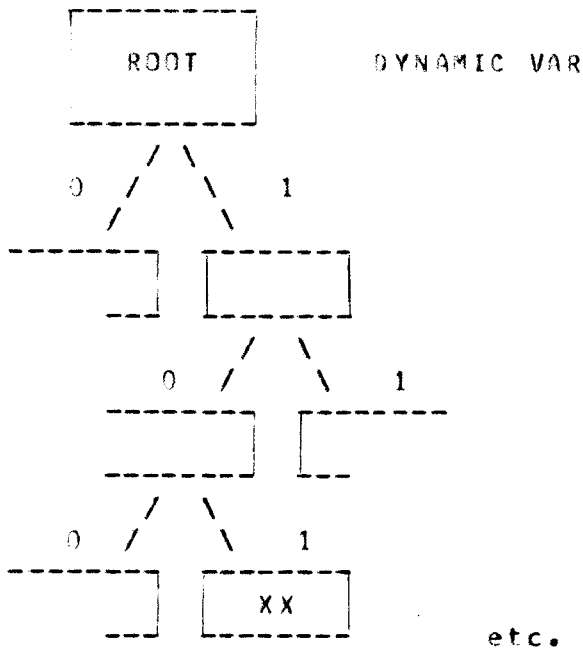
Dynamic variables are those whose values have been altered at some time in a program prior to their use in the current instruction. These variables are the crux of the arbitration problem between competing processes when it comes to a need to refference or update a certain variable. Dynamic variables have at some time appeared as the leftmost operand (op1) in an instruction. It is no longer safe for use without some form of further checking as to it's status in memory.

It is worth noting that a variable may be static up to a certain point in a program, at which it is used as a left-hand operand and becomes dynamic. Thus, some overhead will be conceded for the purpose of maintaining the true current status of a variable, and redundancies will be avoided. This updating will occur as changes happen, rather than as the changes become signifigant. For instance, a variable's status will be changed from static to dynamic as it is used as op1, rather than later when it is requested for use as an op2 or op3.

The method for keeping track of whether a variable is currently static or dynamic (as well as other status information) is the use of ordered binary tree structures for the organization of this information. The desired node of a particular tree is indexed by a sequence of binary digits indicating a traversal from the root node. A "1" in the string indicates a traversal to the right child node, a "0" the left. Since the method is formalized as such, a great deal of time would be saved in letting this be a hardware executable traversal. For the sake of simplicity, it is assumed to be as such.

To reference an operand is to access the key for the tree in which its status indicators reside. Since there are two distinct variable types, there are two distinct trees; a Static Variable Tree and a Dynamic Variable Tree. The distinction is made since the information required for a static variable is different than for a dynamic variable. So, for a variable X with tree location "Treeloc" (0100110), the string would be read right to left, with the rightmost bit indicating the tree (0 -> static, 1-> dynamic) and the remaining bits indicating the right-left branches to be taken in the traversal. An n-bit Treeloc indicates a tree of depth n-1, since the root node is not used for variables. As the number of variables in a given program increase, so increases the legnth of the Treeloc index and the size of the respective tree.

For example, the Treeloc index (1011) will refference node XX.



The Static Variable Tree is structured similarly. In fact, the only distinction lies in the format of their respective nodes.

The format of a node of the Static Variable Tree is as follows:

current value
location of last occurrance

In keeping with the updating of the trees, a variable's last use in a program (as any operand) is noted during the control processor's pass over the code. After this point is reached in the program, it's node is logically deleted from the tree to cut down on unused nodes in the tree.

The format of a node in the Dynamic Variable Tree is as follows:

current value
location of last occurrance
location of last instance as op1
current PSL # for last instance as op1
loop variable flag

Another distinction to be made is that of node entries which are set up to be modified during the control processor's first pass (ie the location of last occurrance) as opposed to node entries which are designed to be used and updated during execution. Current value, location of last instance as op1 (relative to the current instruction) and the current PEL # (relative to the current instruction -- more later) are execution dependant node entries. Current value indicates the variable's present value when it is being used as an operand in the current instruction, and location of last instance as op1 indicates the most recent change of that variable before the present instruction. The current PEL # referrs to the indivisable block of code which contains the instruction that updated the variable's value.

CODE HANDLING CONVENTIONS:

Consider the following sample code;

```
          .
          .
          .
 1    LOAD   A,0          set A to zero
 2    LOAD   B,1          set B to 1
 3    LOAD   C,29         set C to 29
 4    LOAD   D,3          set D to 3

 5    LOAD   A,D          A = 3
 6    MULT   A,A,C        A = (3*29)
 7    ADD    A,A,C        A = (3*29)+29
 8    SUB    C,C,A        C = 29-(3*29)+29

 9-> SUB    D,D,B        D = (3-1)

10    MULT   D,D,A        D = (3-1)*((3*29)+29)
11    SUB    D,D,C        D = (3-1)*((3*29)+29)-(29-((3*29)+29))
          .
          .
          .
```

This somewhat meaningless example demonstrates a potential characteristic of sequential code; it is heavily sequence dependant. That is, most of the instructions are required to be executed in the given sequence in order to get any meaningful result. In fact, there is only one instruction in the

example which may be executed at any time prior to its location in the sequential code, this being indicated by the arrow. The remaining code must be executed in the given order since

in line 11, D depends on line 10 and
C depends on line 8
in line 10, B depends on line 9 and
A depends on line 8
in line 8, A depends on line 7
in line 7, A depends on line 6
in line 6, A depends on line 5.

This is the problem raised by dynamic variables. The advantage of static variables becomes readily apparent, as any static variable has no dependancies and if an instruction has static op2 and op3, it can be executed at any time prior to its occurrance in the original code sequence.

At most, this example, then, could be executed concurrently as two distinct blocks of code, one block including line 9 and the other including everything else. Unfortunately, this speeds things up by only about 16%, which is an insufficient return of efficiency to justify the given model. This brings about the point that there are certain restrictions on how much a piece of code can be altered to be executed in parallel. Ultimately, an ideal interaction between programmer and machine would require the former to have some knowledge of the nature of the system in order to aid in setting up code that lends itself to concurrent processing to some degree. However, in this model, that will not be considered at the surface, since a reasonable return in efficiency is gained when "typical" sequential algorithms are considered.

Thus, the characteristic which makes instruction 9 so different than the other instructions is that both of its right-hand operands are currently static when it occurrs. This hunting for static right-hand operands becomes the main approach in handling sequential code, especially in simple sequence form. The actual mechanism for partitioning code is discussed later. For now, suffice it to say that code is partitioned into the minimum possible segment of a sequence of instructions which require sequential execution.

In handling the more complicated code structures, certain tradeoffs are taken. For conditional code (ie, IF - THEN structures), the approach used in this model is to handle the possible alternate segments of code as regular

sequences of instructions. By regular programming conventions, these alternative branches will either be used or not used, and are thereby distinct. In this light, the control processor is allowed to run right through decision structures, sectioning off blocks of code without actually knowing the outcome of any comparisons evaluated at execution time. Each separate block of conditional code will begin a new block, and therein lies the trade-off. If the outcome of the comparison was known during the creation of these blocks, some efficiency could be gained in some cases. However, this defeats the purpose of doing this partitioning prior to execution, and in doing so, the overall gain in efficiency is maintained.

The handling of loop structures poses more signifigant problems. Loop structures can be broken down into two types; (a) those loops which do not rely in any way upon the outcome of any previous iterations, and (b) those loops which require the previous iteration to be completed before the present iteration can take place. Loops of type (a) will be referred to as "iteratively independant" and loops of type (b) as "iterativley dependant."

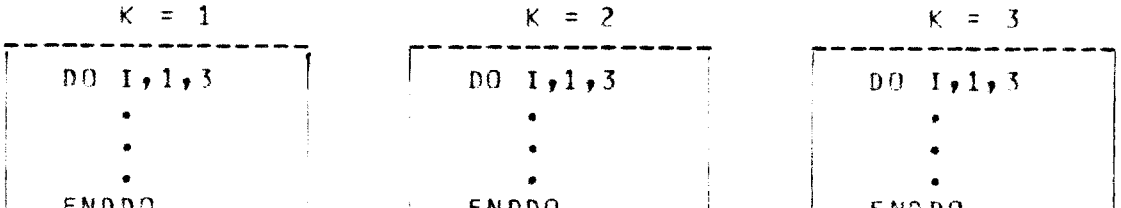
The ideal way of handling a loop structure is to cause all of its iterations to be executed simultaneously, thus making full use of whatever available processing resources are at hand. However, loops of an iteratively dependant nature deterr these concurrent considerations. If a loop requires that the (n-1)th iteration be completed before the (n)th iteration can take place, these requirements must be satisfied. All that can be done is to take any measures possible to make the code within the loop as efficient as possible. It would seem that iteratively dependant loops will always be at odds with concurrent processing, and may one day be replaced by techniques more suited to parallelism. This is not presently the case.

Iteratively independant loops, on the other hand, provide a great deal more in the way of parallelizable options. Consider the code designed to find the product of two (3 x 3) matrices;

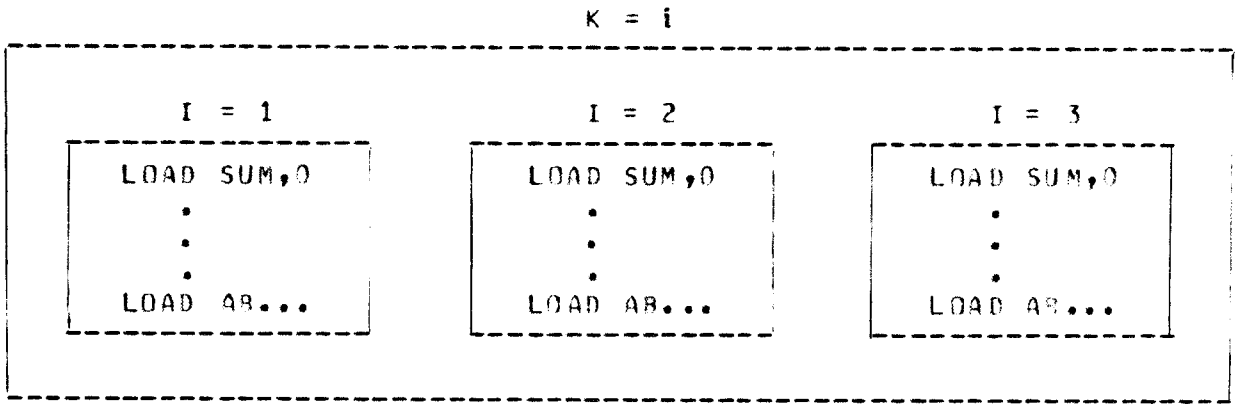
```
DO    K,1,3
   DO   I,1,3
      LOAD SUM,0                          set sum = 0
      DO   J,1,3
         MULT ATEMP,A(I,J),B(J,I)
         ADD  SUM,SUM,ATEMP
      ENDDO
      LOAD A(K,I),SUM                     move product to matrix A
```

```
      ENDDO
ENDDO
```

Since the running sum SUM requires a value from the previous iteration, the inner loop J is ineratively dependant. However, there is no such restraint for the outer loop K, which could be broken down into its respective iterations, each iteration operating for a single vaue of K to be determined at execution time. Hence, a safe partitioning of the three iterations of the outer loop K could be



At this point, the algorithm will take only 1/3 its previous execution time. However, since the I loop is iteratively independant, provided it has a value for K in it's last instruction, the I iterations can be broken down similarly within each concurrent iteration of the K loop. Thus, for K(i),



where any refference within the body of an I iteration to the loop variable K would be replaced with the "hard" value of K for that K loop's relavent iteration. In this way, the algorithm now runs at 1/9 of its original time, provided there are at least nine processors to handle the iterations.

The measures which would need to be taken to enable the inner loop to run concurrently with its own iterations would require more overhead than is feasible for the situation. If the value of SUM could be determined

for each iteration of the innermost J loop, then added with all the other "SUM" values from the other respective concurrent J iterations within each iteration of the I loop, the general speed of the algorithm could be again squared. The cost of such specialized measures, however, make such an action rather awkward, and the overhead necessary to facilitate such a design would undermine the simplicity and efficiency of the model. Thus, loops will be broken down as much as possible within the conventions demonstrated in this example.

SPECIFIC METHODS OF CODE HANDLING

Actual considerations for the partitioning of sequential code are now dealt with in detail. Consider the concept of a Process Extent List (PEL) which is defined as a minimum block of code which must be executed in its original sequence. As the Control Processor makes its pass over the code, it generates the object codes for the instructions. A PEL is made up of groups of these object codes.

Object codes for the general set of instructions (of form NAME op1, op2,op3) are laid out as follows;

where

field 1 -- op code of the instruction
field 2 -- current PEL #
field 3 -- pointer to Treeloc Table for op1
field 4 -- PEL # in which op2 was last altered
field 5 -- pointer to Treeloc Table for op2
field 6 -- PEL # in which op3 was last altered
field 7 -- pointer to treeloc table for op3

PEL's are numbered according to the their creation based on the original sequence of the input code. Therefore, the current PEL # referrs to the current PEL in which this current instruction resides. Since it is necessary to keep track of when an operand was last altered (if it is a dynamic operand), the PEL # referring to that instance is included in the object form of an instruction. Treeloc Table index referrs to the table created during the first pass that associates a variable with a Treeloc, and thereby a Static or Dynamic Variable Tree node.

Two operand instructions (ie LOAD op1,op2) have similar object code formats to three operand instructions, and merely lack the information concerning operand 3. The format for loop instructions is also roughly equivalent to that for standard three-operand instructions, except that

a small flag is included to indicate the actual loop variable. This flag may also reside in the object descriptions of normal operands, and this fact becomes of prime importance when the mechanism for determining loop iteration dependancy is examined.

The importance of introducing operands' PEL as in instruction codes becomes apparent in considerations of the execution priorities of PELs. The convention is that the current instruction may not be exected until the PELs indicated for its operands have been completely processed. This is a result of the fact that only dynamic variable will have PELs indicated in the instruction code, and dynamic variables are the primary cause of the arbitration/priority problem.

The basis for creation of PELs becomes more clearly defined when the Control Processor's initial pass over the sequential code is examined. In the pass algorithm, it is seen how PELs are created, what conditions are required to terminate the continuation of the current PEL, and what criteria are to be met for a new PEL to be created. The general rule of thumb is that as soon as a new PEL is creatable, the current PEL should be discontinued. The emphasis is on keepin PELs as short as possible, since each one represents an indivisable block of sequential code. For example, a program running within a single PEL is a sequential program. The greater the number of PELs, the greater chance that more than one can be executed at any given time, and thus the greater the overall efficiency of the program in execution.

CONVENTIONS FOR DETERMINING DEPENDANCIES OF DYNAMIC VARIABLES

For general sequential code, it is enough (as implied by the treatment of instruction formats) to indicate which, if any, PEL a particular variable depends on. So, for general code, an operand depends on a previous PEL

if (1) it is not a new "literal" constant
(2) it is not currently static
(3) it does not depend on an op1 in any previous PELs.

The method for determining if an operand depends on an op1 in any previous PELs throws back to the node contents of the Dynamic Variable Tree. During execution, the last instance of that variable as op1 in any instruction is noted and recorded in the DVT node. Also noted is the corresponding PEL # for that instruction, and by refferencing this, the PEL in which the inspected variable last occurred as op1 in an instruction is obtained. This method allows the system to look back to only the last change of the variable ("last" in the context of the original sequence of the code) rather than backtracking farther than is actually necessary.

With regard to dependancies of loop variables, the loop flag comes into significance. In the code

```
DO    K,1,10
   ADD   I,J,K
   MULT A,I,B
   LOAD C,H(A,N)
ENDDO
```

K is the loop counter, and its DVT entry has associated with it a Loop Variable flag. Since K appears as a right-hand operand in the ADD instruction, the variable I now depends on K. Since I appears on the right-hand side of the MULT instruction, A depends on I which depends on K. Since A appears as a subscript in the array refference on the right-hand side of the LOAD instruction, C now depends on A.... Since all of these variables are indirectly dependant upon the loop variable K, they are all unable to assume the roles of independant variables within the context of

the loop K or in any subsequent loop residing within the K loop.

If there was no way of indicating a variable's dependancy upon a loop variable, there would be no safe way of checking for iterative dependance or normal dependance within loops. This being the case, the following convention is established: If there is a loop counter K, a flag indicative of this is included in its DVT entry. Furthermore, any variable which depends on an operand with this flag set will also have its own flag set in its DVT entry. It is for this reason that the refference to an variable's last instance as op1 in an instruction is included with that instruction's PEL # in the DVT entry. The actual instruction must be refferenced to check its operands for loop variable dependancy.

Before fully revealing the nature of the Control Processor's initial pass over the sequential code, the algorithm for handling loop structures will be examined. This is part of execution procedure and does not take place during the control processor's first pass.

Consider a system configuration with N processors, and an instance within the code of a loop structure consisting of n iterations. Before execution time, N is known, but n may or may not be. It is for this reason that the actual loop processing takes place at execution time. The algorithm for dealing with loop structures during the execution cycle is as follows:

```
If instruction is of form ( DO   K,A,B )
  If loop is iteratively independant
    n = B - A
    If n > N
      Treat the body of the loop sequentially
      Concurrently process loop iterations (A,A+1,...C-1,C)
                                        where ( A < C < B )
      Update A = C + 1
      Transfer control to top of execution cycle for that instruction
    Else
      Concurrently process loop iterations (A,A+1,...B-1,B)
      N1 = N - n
      Transfer control to top of execution cycle for loop body with
                                    N1 available processors.
    Endif
  Else
    Execute loop iterations in order
  Endif
Endif
```

This executions step allows for as many loop iterations of an iteratively independant loop to be executed concurrently as there are available processors. If the loop can be handled and processors remain, those may be allocated otherwise. If there are two few processors to handle all loop iterations at once, all processors are dedicated to processing the loop until the remaining iterations are depleted. Recursion is a suitable method of implementing the transfer of command to the top of the fetch execute cycle, although the actual method is not considered in detail.

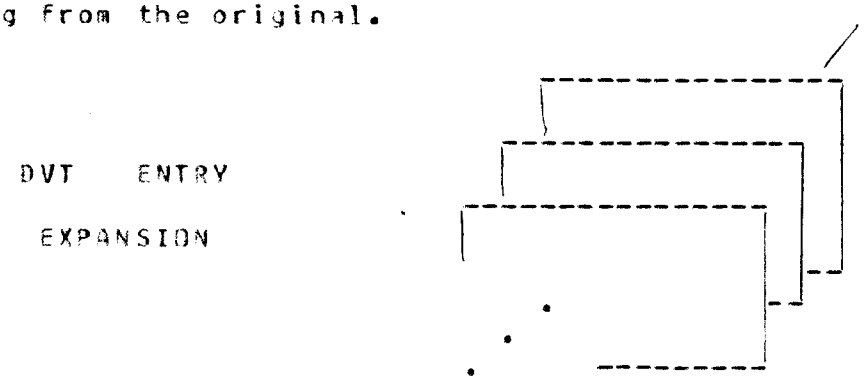
In as much as most everything else regarding the partitionability of the sequential code can be deduced prior to execution, the Control Processor's initial pass determines all other details required by the model. In this passes' handling of loop structures, the assumption is made that a loop will be of the worst possible form, and, although notes are made of loop variables and their dependants in the DVT, the code within loops is treated as basic sequential code and handled like the rest of the program.

Since, in the case of iteratively independant loops, the method of handling in this model is to partition the loop into as many separate iteratins as are possible, some consideration must be given to handling variables within the loops themselves. Static variables are harmless, being of a "read only" type, the arbitration of which would be dictated by hardware. However, dynamic variables changing within a loop cause problems. N parallel instances of a dynamic variable in a loop will create N potentially different values for that variable, which must ultimately be resolved into one.

The case of a dynamic variable accumulating its value through successive iterations of a loop will not be a problem here, since such a condition constitutes an iteratively dependant loop, which, by the above algorithm, will not be partitioned. This simplifies the task of handling dynamic loop variables tremendously.

The general method for breaking up a loop into its iteration is to

expand upon the DVT entries for dynamic variables appearing within that loop. Copies are made of the DVT entry, and linked outward to such an extent as is dictated by the above algorithm. For N concurrent loop iterations, N-1 linked copies of the variable's DVT entry will be created, stemming from the original.



As indicated before, the loop counter's values for respective loop iterations are fixed to each iteration. The code of the loop itself is not copied and is here assummed to be simultaneously useable by all available processors. Variables chosen for "duplication" must not be subscripted. If a subscripted variable appears on the left side of an instruction, and if that variable (with any subscript at all) appears on the right side of an instruction, the loop is non-parallelizable, as it has implied potential iterative dependance. What this boils down to is that any subscripted variable which is dynamic within the loop cannot be a right-hand operator. This concept of being static/dynamic within a loop is based on the fact that a dynamic variable can appear to be static if considered only within the bounds of a loop structure. The model exploits this fact to wring a bit more efficiency out of loop handling.

The way in which the copies of the DVT entries are used is to associate each available processor with a particular DVT entry copy. Whenever a processor processing a particular iteration refferences a particular dynamic variable (with multiple DVT entries), it sees only a particular DVT-entry-incarnation. In this manner, these variables may be used and altered independantly. The manner of resolution of final value after the loop is processed is to move the status of the variable corresponding to the loop iteration with the highest value (assuming no direct decrementation

copies are then deleted. The number of copies will rarely reach an intolerable number since it is directly related to the number of iterations to be partitioned at a time, which in turn is directly related to the number of processors available. If the size of the main memory is at all sufficient for the needs of N processors, this demand will not exceed the capabilities of the system. Thus, an adequate hardware structure will not be taxed by this method, providing that it was constructed realistically for the number of processors it contains.

THE FIRST PASS ALGORITHM

```
Initialize instruction counter
Initialize Treeloc indexes as ( null )     ( DVT and SVT roots )
Initialize PEL #  = 0
look at first instruction
(top of loop)
Determine appropriate object code form
Move associated op code to object form's first field
Move current PEL # to object form's current PEL # field
Associate operands with object form's operand entries

If ( DO )
  Examine op1
    Set up a DVT entry          (assumed to be unique and dynamic)
      Move variable name to index name field
      Move next tree index to Treeloc field ( for DVT )
    Refference indicated DVT entry by Treeloc
      Move loop variable-flag to DVT entry  ( assert flag )
      Move current instr. cntr. value to last-changed-field of DVT entry

Else if ( COND )
      Create a new PEL
        Increment current PEL #
        Move this new PEL # to curr. PEL # field of instruction's obj form

Else
  Examine op1
    If ( no duplicate DVT entry exists )
      If ( duplicate SVT entry )
        Delete that SVT entry and its index
      Endif
          Move variable name to index name field
          Move next tree index to Treeloc field ( for DVT )
    Endif
      Refference indicated DVT entry by Treeloc
        Move curr. instr. cnt. to last change field of DVT entry

  Examine op2
    If ( not already defined in DVT )
      If ( no duplicate SVT entry )
        Set up SVT entry
          Move variable name to SVT index
          Move next tree index to Treeloc field ( for SVT )
      Endif
    Endif
    Move curr. instr. cnt. to last occurred field of tree entry
    If ( loop-variable flag is on )
      Set loop-variable flag in DVT entry for op1
    Endif

  Examine op3
    If ( not already defined in DVT )
      If (no duplicate SVT entry )
        Set up SVT entry
          Move variable name to SVT index
```