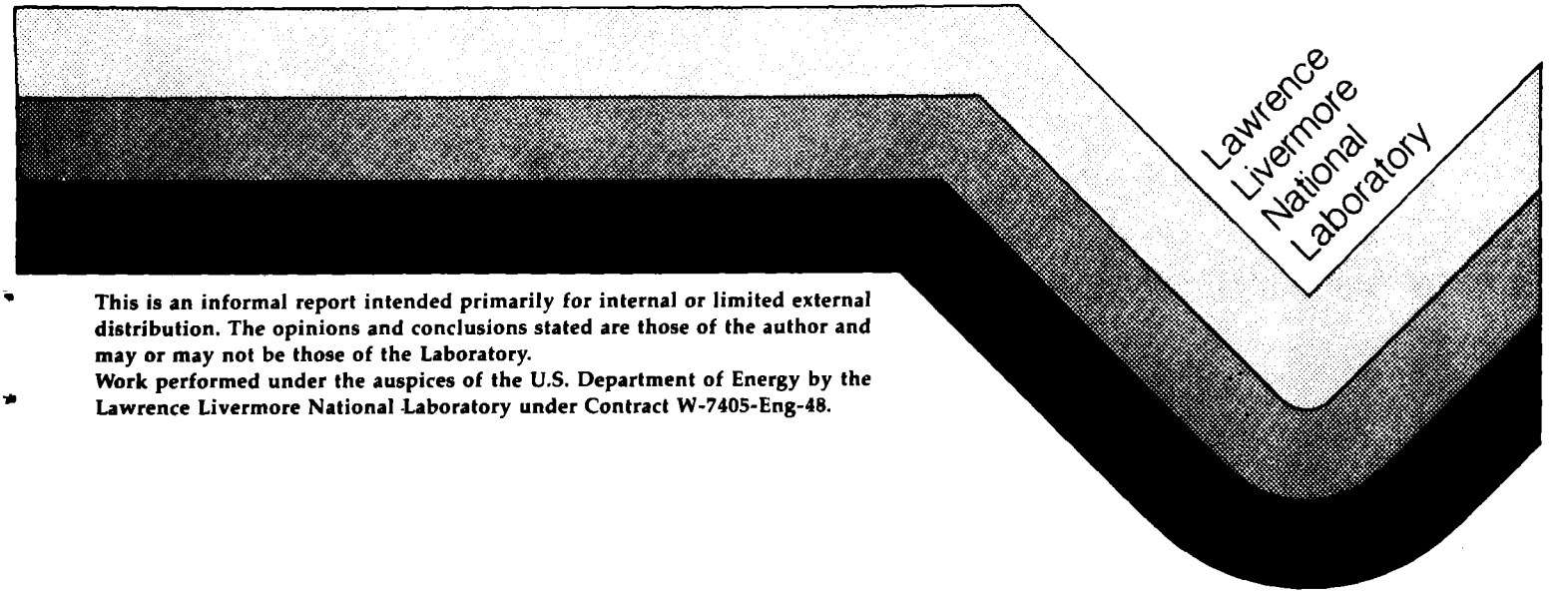
UCID-21159

THE LIVERMORE LOOPS IN SISAL

John T. Feo

CIRCULATION COPY
SUBJECT TO RECALL
IN TWO WEEKS

August 26, 1987



This is an informal report intended primarily for internal or limited external distribution. The opinions and conclusions stated are those of the author and may or may not be those of the Laboratory.
Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

Printed in the United States of America
Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161

Price Code	Page Range
A01	Microfiche
Papercopy Prices	
A02	001 - 050
A03	051 - 100
A04	101 - 200
A05	201 - 300
A06	301 - 400
A07	401 - 500
A08	501 - 600
A09	601

The Livermore Loops in Sisal

John T. Feo
Computing Research Group
Lawrence Livermore National Laboratory

Abstract: The Livermore Loops are 24 computational kernels found frequently in large-scale scientific computations. Sisal is a functional language being developed at Lawrence Livermore National Laboratory, to provide a programming medium in which to express large-scale scientific processes for execution on multiprocessor systems comprising hundreds, even thousands, of processors. For Sisal to be successful, it must express clearly and concisely the parallelism found most frequently in scientific computations. We present and discuss a Sisal implementation of the Loops from both a programming and execution viewpoint. In addition, we give the parallel execution traces of all 24 loops as simulated by our interpreter.

Author's Address: Lawrence Livermore National Laboratory (L-306), P.O. Box 808, Livermore, CA 94550

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Department of Energy, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately-owned rights.

References to a company or product name does not imply approval or recommendation of the product by the University of California or the United States Department of Energy to the exclusion of others than may be suitable.

1.0 Introduction

The Livermore Loops [8] are 24 Fortran loops from actual production codes run at Lawrence Livermore National Laboratory (LLNL). The Loops represent the type of computation kernels typically found in large-scale scientific computing. They range from common mathematical operations, such as inner product and matrix multiplication, to intricate searching and storing algorithms, such as the Monte Carlo Search Loop and the 2-D Particle-in-Cell Loop. The Loops' computational structures are also well varied, including: massively parallel processes, recurrent processes, wavefronts, and pipelines. The Loops provide an excellent testbed to evaluate the appropriateness and expressive power of parallel languages.

Sisal [7] is a functional language being developed at LLNL for large-scale scientific processing. The goal is to provide scientists with an easy-to-use parallel programming language in which to express scientific computations for execution on large-scale multiprocessor systems. Sisal executes currently on the Manchester Data Flow Machine [4], the MultiVax [10], and the Sequent Balance 21000 [10]. A native code compile for the Cray X/MP-4 will be available soon.

All parallel languages have either implicit or explicit constructs to express the parallel nature of algorithms, and the communications and synchronization inherent in parallel processing. In functional languages the constructs are all implicit. From the user's prospective, programs are sets of high-level expressions that are compiled to graphs whose nodes are operations and whose edges are data. A node is executed as soon as all of its inputs are present. The order of execution, and thus the attainable parallelism, is limited only by data dependencies and the availability of results. The flow of values and the firing of ready nodes is the responsibility of the run-time system. Users need not, in fact can not, mange the scheduling or synchronization of the parallel units of computation within their applications. They must leave such operational details to the compiler and the run-time system. Functional languages can provide the easy-to-use parallel programming environment we seek.

But if Sisal is to be an appropriate language for large-scale scientific computing, it must do more than isolate scientists from the vicissitudes of parallel processing – it must also express clearly and concisely the parallelism within their computations. A parallel language defines a set of abstract machines. Applications written in the language are mapped to these machines [2]. If the mapping is awkward or in any way obfuscates the algorithm's parallelism, the abstract machine (i.e., the language) is inappropriate for that type of process.

This report evaluates the appropriateness of Sisal as a parallel language for scientific computations by writing the 24 Livermore Loops in the language. The evaluation metric is the complexity of the Sisal function compared to the complexity of the underlying mathematical computation as reported in [3]. Complexity is defined as the execution time of an algorithm as a function of its input size. An algorithm is $O(n)$ if it completes in n logical steps, where a step includes all operations that can execute at a particular instant of time. Figure 1 depicts a binary tree computation of $O(Log\ n)$. The input size is 16 and the number of logical steps is 4. The *width* of the graph is the order of the computation and the *height* of each step is the number of operations that can execute concurrently at that time. The appendices include the execution traces of the Sisal functions as simulated by our interpreter.

The second section describes the general nature of the study and the current Sisal programming environment at the Laboratory. The reader is assumed to have a working knowledge of Sisal; if not, see [7]. Section 3 discusses a Sisal implementation for each of the 24 loops, and section 4 presents our conclusions. Appendix A and B give the Sisal and Fortran versions of the Loops, respectively, and Appendix C gives the parallel execution profiles of the Sisal functions as simulated by our interpreter.

2.0 General Considerations

The Sisal functions faithfully implement the computations of the Loops; as executed by our interpreter, their results are identical to the Fortran routines. However, no effort is made to return the same structures as those returned by the Fortran loops. Because Sisal is a single-assignment language, updating arrays is an unnatural concept. If a Loop replaces only certain elements of an array, the Sisal function computes the new values but may not return the entire array; instead it returns only an array of the new values. For example Loop 8 recomputes the subarray *U1(2..3, 2..n, 2)*, the Sisal function returns only the new values and not the entire *U1* array. To affix the unchanged elements of *U1* to the new values is awkward in Sisal because of a lack of syntax.

We realized early that direct Fortran to Sisal translations are inappropriate. Except for the most massively parallel loops, such translations are difficult to understand and not optimal. This finding is profound. It implies that the formulation of efficient parallel algorithms for scientific computations must begin at the original mathematics or physics of the problem and not at the implementation of the computation in some conventional language. Translating codes from one language to another generates a composite mapping that may introduce constraints that are not otherwise present in a map from the original mathematics to the target language. This is particularly

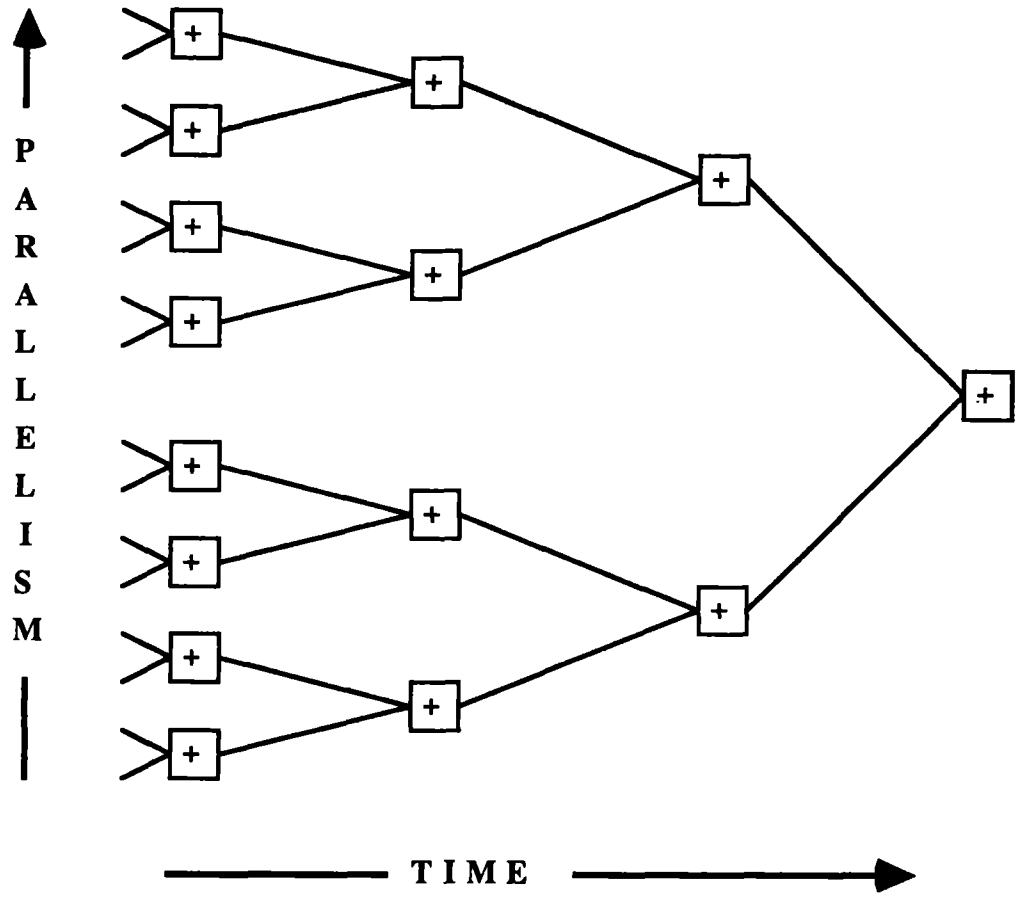


Figure 1: An O(Log n) Binary Tree Algorithm

true if the first language is sequential and the second is parallel, since the former may introduce unnecessary sequential constraints.

The Sisal programming environment at the Lawrence Livermore National Laboratory consists currently of a compiler, a debugger/interpreter, a profiler, and other tools. Sisal programs are written using any standard text editor and compiled. The compilation process transforms programs to an intermediate graph form known as IF1 [11], and can correct a variety of syntax errors and perform a number optimizations, including: common subexpression removal, loop invariant removal, loop test inversion, and constant folding [12]. IF2, a second intermediate form, is under development and will include: vectorization, update-in-place, and preallocation of memory [14]. DI, the debugger/interpreter executes a sequential depth-first traversal of IF1 graphs [13] and

if successful, returns their results. In the debugger, users can set breakpoints, view the values on edges, trace function calls, save the results of functions for later use, and request a trace of executed nodes.

The trace of executed nodes lists each node's unique identification number, its type, and the identification numbers of its input producers. The Profiler reads the listing and generates a histogram of the start-end times of the nodes. Start time is defined as the maximum of the end times of the input producers and any boolean expression on which the node's execution depends, plus communication delay. End time is defined as start time plus the execution length of the operation. For simplicity we assume the communication delay between any two operations is 0 and the execution time of any operation is 1 (including noops). Library functions appear in IF1 graphs as single nodes, so all complete in a single time unit regardless of complexity. The Profiler respects all data and logical dependencies. It implements both loops and **if-then-else** constructs strictly; i.e., no loop value is available until the entire loop finishes, and neither branch of an **if-then-else** statement can execute until all the inputs to both branches are available.

Appendix C gives the histograms of the execution traces of the 24 loops as generated by DI and the Profiler. The graphs are included to illustrate the parallel nature and order of the routines, and not to give an exact log of the executed instructions. In some cases the order of the trace is incorrect because DI simulates the reduction of parallel loops as a set of concurrent operations. For example, if the results of a parallel loop of size n are summed, DI simulates the reduction as n concurrent sums. The Profiler, thus, accumulates all the operations in one bin and the reduction appears to be *O(1)* when it is at best *O(Log n)*. This is referred to as the *reduction anomaly*.

3.0 The Loops

3.1 Loop 1

Loop 1 (Appendix A, page 34; Appendix C, page 56) returns an one-dimensional array of size n. The i^{th} element is set to

$$Q + [Y(k) * [R * Z(k+10) + T * Z(k+11)]] .$$

A **for all** expression computes the values in parallel and gathers them into an array. The Sisal function is *O(1)* and optimal. [We will refer to parallel loops as **for all** expressions and to iterative loops as **for** expressions.]

The trace of Loop 1 *(n = 990)* is easily analyzed. At time 0, 990 instances of the body are scattered. At step 1, three instructions execute per loop body for a total of 2970 operations; they are: the array access *Y(k)*, the sum $k + 10$, and the sum $k + 11$. At time 2, the *Z* array is accessed twice (1980 operations); and at time 3, the multiplications with respect to *R* and T execute (1980 operations). The inner addition, the multiplication by *Y(k)*, and the addition of *Q* occur at steps 4, 5, and 6, respectively, for a total of 990 operations per step. Finally at time 7, the interpreter gathers the 990 results into an array; and at time 8, sets the array's lower bound to 1.

3.2 Loop 2

Loop 2 (Appendix A, page 34; Appendix C, page 57) builds a one-dimensional array of approximately $2n$ elements in $\lfloor Log\ n \rfloor$ steps. Step i, $0 \le i \le \lfloor Log\ n \rfloor$, computes the next $n/2^i$ elements. The steps must execute sequentially since the elements computed at step i are functions of elements computed at step $i - 1$; however, the computations of an individual step are data independent and can execute in parallel with one exception. If $n/2^{i-1}$ is even, the last element of step i is dependent on the first element of the step [3] and the two computations must be ordered.

The Sisal implementation comprises an outer sequential loop and an inner parallel loop. The sequential loop executes $\lfloor Log\ n \rfloor$ times and concatenates the arrays returned by the **array_addh** operation. Concatenation is a reduction operation provided by the language. To make the code easier to understand the last element is calculated separately whether or not it is dependent on the first. Since Sisal arrays are dynamic, the varying size of the component arrays returned by the inner loop is handled naturally. The **old** construct gives access to elements computed on the previous iteration and makes possible the piecemeal construction of the returned array. Without the construct, each iteration would have to append its elements to those computed previously and return the entire structure. Such an operation in a single-assignment language is extremely inefficient as it requires copying the previous results – although in this case, simple analysis could uncover the redundancy and eliminate it.

The Sisal implementation is clearly *O(Log n)*. The execution trace in Appendix C for $n = 101$ has 6, or $\lfloor Log\ n \rfloor$, segments of constant width (18 steps). The height of each segment is directly proportional to the parallelism of the inner loop; since each step computes half as many elements as the previous step, the heights fall off logarithmically. Much of the evaluation of the last element executes concurrently with the inner loop; but, due to the strictness of the **if-then-else** expression neither branch of the **if** expression can execute until the inner loop completes. This serializes the final steps in the evaluation of the last element and produces the wide gaps between segments.

3.3 Loop 3

Loop 3 (Appendix A, page 34; Appendix C, page 58) calculates the inner product of $\underline{X}$ and $\underline{Y}$. The partial products are computed in parallel in a **for all** expression and summed by the **value of sum** reduction operator provided by the language. The function is *O(Log n)*. The execution trace exhibits the reduction anomaly.

3.4 Loop 4

Loop 4 (Appendix A, page 35; Appendix C, page 59) returns the array *X* with element *X(k), k = {6, 503, 1000}*, replaced by

$$Y(5) * \left[X(k) - \sum_{i=1}^{n/5} [X(k-6+i) * Y(5*i)]\right] .$$

Since $\underline{X}$ is dimensioned 1001, *n* must be less than or equal to 35 – otherwise, the calculation of *X(1000)* would go out of range. Since *n* is so restricted, the three computations are disjoint in $\underline{X}$ and can execute in parallel. The *X(k)* term appearing explicitly in the equation refers to an old value and is of no consequence. However, a *X(k)* term appearing in the sum refers to the interim value of *X(k)* at $i = 5$. If it should appear in the sum (and it will whenever $n \geq 30$), we must replace it by the old value of *X(k)* minus the accumulated sum up to that point.

The Sisal implementation calculates the three values in parallel in two instructions. The first calculates the terms in the sums for *i* less than *6* – it comprises three **for all** expressions embedded in a multi-expression. The second instruction is a three branch **if-then-else** expression that calculates either no additional terms, a sixth term, or a sixth and seventh term. The second and third branches are actually a nested **if-then-else** expression within the else arm of the first expression. Because **if-then-else** expressions are strict, the interpreter can not execute any part of the second expression, including the test, until all inputs to both arms are ready. This delay causes the gap in the execution trace from time 8 to 11. For a detail explanation of how DI implements **if-then-else** expressions see Section 3.22. The function in Sisal is *O(Log n)* and optimal. The first feature of the execution trace suffers the reduction anomaly.

3.5 Loop 5

Loop 5 (Appendix A, page 35; Appendix C, page 60) returns the array *X* with *X(i), i = {2..n}*, set to

$$Z(i) * [Y(i) - X(i-1)] .$$

The equation is obviously recurrent, but the data dependency can be eliminated by substituting recursively for *X(i-1)*. For example, the equation for *X(5)* can be written as

$$Z_5Y_5 - Z_5Z_4Y_4 + Z_5Z_4Z_3Y_3 - Z_5Z_4Z_3Z_2Y_2 + Z_5Z_4Z_3Z_2X_1 .$$

Unrolling the equations reduces the execution time of the loop from n to *Log n*, but increases the number of operations for n to n^3. Because of the explosion in the number of operations, the parallel version is likely to be more expensive than the sequential version.

There is a third alternative. By executing the operations in a structured manner and saving partial results, we can reduce the loop's execution time to *Log n* while increasing the number of operations to only *O(n Log n)*. The method is known as recursive doubling and is described in [5, 6]. Figure 2a shows the computation graph for Loop 5 using this technique. At step 0, there are n computations and the computation of the first element of $\underline{X}$ completes. At step i, $1 \le i \le \lceil Log\ n \rceil$, there are $n - 2^{i-1}$ computations and the evaluation of 2^{i-1} elements of $\underline{X}$ complete. Computation j of step i uses the results of computation j and $j - 2^{i-1}$ of step $i - 1$. Each computation returns two results: a product of *Z*s and a partial value of an element of $\underline{X}$. Figure 2b shows the computation subgraph for *X(5)*.

The Sisal code for this alternative is given in Appendix A. The code is clean and concise; it is almost an exact description of the computation graph, there are no spurious statements. Each iteration of the **for** expression returns the next row of the graph and the two inner loops return the results of the computations. The **array_adjust** selects those elements in the previous row that have completed. This operation and the associated concatenation could be eliminated if Sisal permitted the replacement of subarrays. Syntax for such operations is being developed.

The run depicted in Appendix C is for $n = 101$. The trace shows the function is *O(Log n)* and the total number of operations (the area under the curve) is *O(n Log n)*. The first feature represents the parallel loops in the initialization clause. Notice the heights of the other peaks fall off logarithmically as expected. Since the elements of $\underline{Z}$ are originally of the order of 10^{-3}, the product of *Z*s underflows on our machine after four iterations. DI sets the underflow to 0 and continues; however, this does not affect the Sisal results – they remain identical to the results of the sequential Fortran routine.

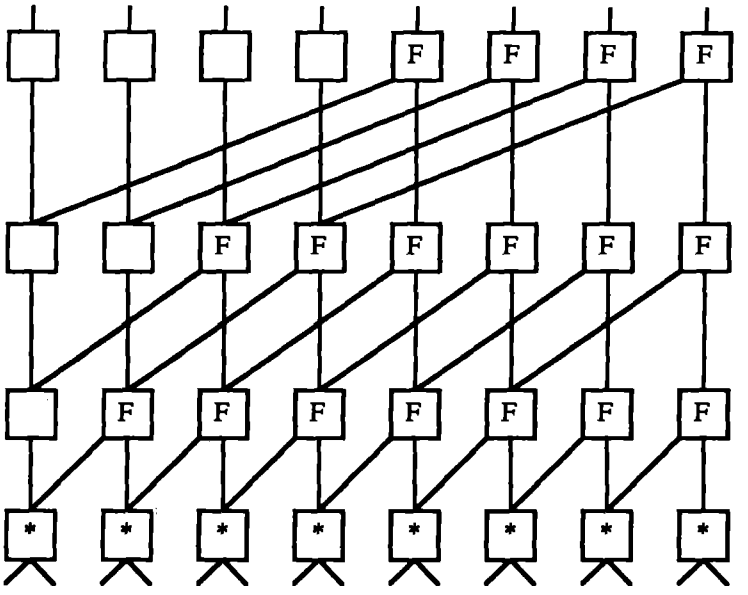


Figure 2a: Parallel Computation Graph Loop 5

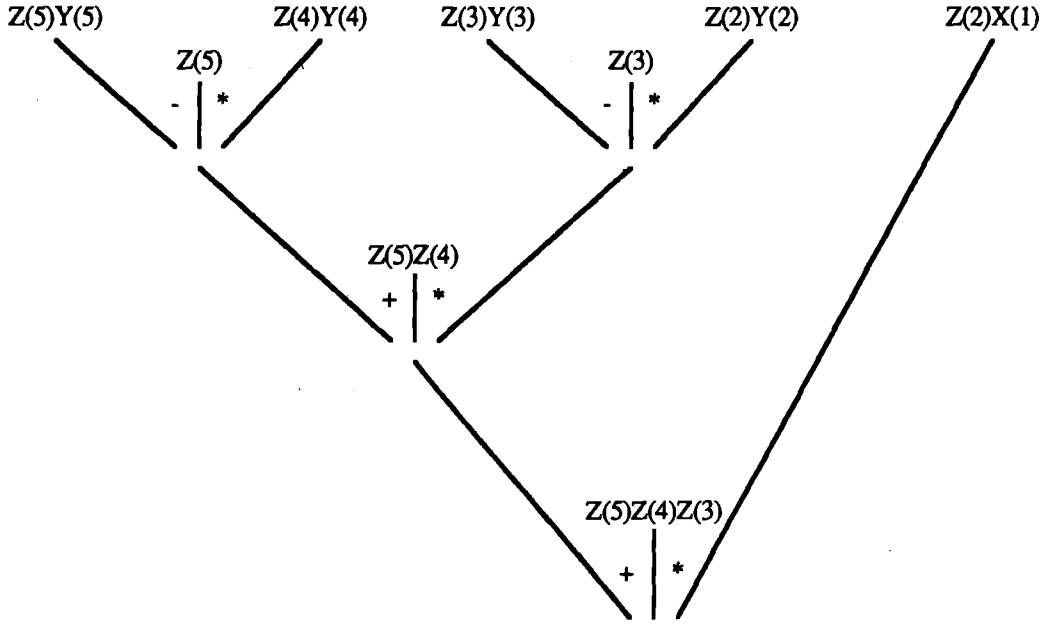


Figure 2b: Parallel Computation Graph for X(5)

3.6 Loop 6

Loop 6 (Appendix A, page 36; Appendix C, page 61) returns the array *W* with *W(i)*, $i = \{2..n\}$, set to

$$W(i) + \sum_{k=1}^{i-1} [B(i,k) * W(i-k)] .$$

An *O(n)* version of the algorithm is described in [3]. At step *i*, *W(i-1)* is multiplied by the appropriate elements of *B* and added to the values of $W(j), j = \{i..n\}$. The calculation of *W(i)* completes at step *i*.

The Sisal implementation consists of an outer sequential loop and an inner parallel loop. On the i^{th} iteration of the outer loop, the parallel loop adds $W(i-1) * B(j, j-i-1)$ to $W(j)$, $1 \leq j \leq n$, and returns an array of incremented values. The returns clause of the sequential loop gathers the first element of these arrays into an array. Since the array access *old W[old i]* is independent of *j*, the optimizer pulls it out of the **for all** and places it in the body of the **for** expression. The value is then passed to the parallel loop as an input parameter. Thus, the array access occurs only once and not $n - i + 1$ times. The execution trace in Appendix C is for $n = 64$. The graph shows the outer loop executes *n* times and each iteration executes in constant time (10 steps); thus the function is *O(n)*. The spikes depict the execution of the inner loops. Their heights decrease linearly since one less sum is computed each time.

3.7 Loop 7

Loop 7 (Appendix A, page 36; Appendix C, page 62) returns a one-dimensional array of size *n*. The i^{th} element of the array is set to

$$U(i) + R * Z(i) + R^2 * Y(i)$$
$$+$$
$$T * [U(i+3) + R * U(i+2) + R^2 * U(i+1)]$$
$$+$$
$$T^2 * [U(i+6) + R * U(i+5) + R^2 * U(i+4)] .$$

The Sisal implementation is *O(1)* and optimal. Because Sisal has no infix operator for exponentiation, we have had to expand the equation in the code. We plan to include such an operator in the next version of the language. Even though the equation is complicated, its fine-grain parallelism is exposed. The number of different instructions executed each step is the height of the

step divided by *n (= 995)*. The quotient is exactly the number of operations that can be logically executed at that time.

3.8 Loop 8

Loop 8 (Appendix A, page 36; Appendix C, page 63) returns three, three-dimensional arrays of size *[2..3, 2..n, 2]*. The Sisal function does not return the same structure as the Fortran code, but only the new values. The three sets of values are computed in parallel by a single, doubly-nested **for all** expression of arity three. The cross product is only notational convenience. Two instances of the loop over *kx* are scattered and each of these scatters *(n - 1)* instances of the body. Since accesses of *U1*, *U2*, and *U3* in the first dimension are independent of *ky*, they are pulled out of the inner loop and executed in the outer loop. Thus, each is executed only twice instead of *2 * (n - 1)* times. Although this optimization does not improve the order of the loop, it does reduce the total number of operations executed.

The execution trace shows the Sisal implementation is *O(1)* and therefore, optimal. The graph for *n = 100* is 18 steps wide and has a maximum height of 2970. The nested structure of the loop and the access of the three input arrays in the first dimension give the graph a sequential component at the beginning and at the end. The middle portion of the graph depicts the parallelism in the body of the loop. Once again, dataflow has exposed the fine-grain parallelism of a complicated function and made it possible for the interpreter to simulate the execution of logically concurrent operations in parallel.

3.9 Loop 9

Loop 9 (Appendix A, page 37; Appendix C, page 64) replaces the first row of *PX* and returns the array. The i^{th} element of the row is set to

```
PX(3,i) + CO * [PX(5,i) + PX(6,i)]
                +
DM22 * PX(7,i) + DM23 * PX(8,i) + DM24 * PX(9,i)
                +
DM25 * PX(10,i) + DM26 * PX(11,i) + DM27 * PX(12,i) + DM28 * PX(13,i) .
```

It is convenient in this case to return the entire structure. Since a two-dimensional array is an array of rows, replacing the first row is equivalent to replacing the first element of a one-dimensional array. What is awkward in Sisal is replacing subarrays which are not natural components. To do

so, the old array and the array of new values must be disassembled and a new array assembled from the parts by concatenation. The difficulty arises because of a deficiency in syntax and is neither intrinsic to the language nor to data flow in general.

The Sisal function for Loop 9 in the appendix is *O(1)* and optimal. One could form a vector out of the scalars and express the equation as an inner product if one wished. The 10 array accesses of PX in the first dimension are independent of *i* and are pulled out of the loop by the optimizer. They execute first creating the blip in the graph at time 0. The terms of the equation are summed from left-to-right giving the graph a long tail. A pair-wise summation of the terms would be both faster and more parallel. Implementing a sum of terms in a tree structure is a worthwhile optimization, but it is not always the most efficient. The time to compute each term and communication delays, which can be known only at run time, may make some other arrangement of terms more efficient.

3.10 Loop 10

Loop 10 (Appendix A, page 37; Appendix C, page 65) returns a two-dimensional array of size *[5..14, 1..n]*. Element *(i, j)* is set to

$$CX(5, j) - \sum_{k=6}^{i} PX(k-1, j) .$$

A doubly-nested **for all** expression computes the instances of the equation and gathers the results into an array. A single expression in the returns clause states the equation. The sum over PX is returned by a third **for all** expression which fetches the values in parallel and reduces them via the **value of sum** reduction operator. The result is then subtracted directly from *CX(5, j)*. The Sisal function is *O(1)* and optimal. The function's complexity is actually *Log i;* but since *i (= 14)* is a constant, the order is 1. The execution trace exhibits the reduction anomaly.

This loop is the best example of how different a Sisal and Fortran implementation can be. The Fortran routine calculates the instances of the equation sequentially and stores the new values back in PX. Since there is an anti-dependency in PX across the iterations, the routine must first save the old PX values before overwriting them. It does this by accumulating the values in three temporaries, *AR, BR,* and *CR*. The introduction of these variables makes it impossible to write the sum as a loop and greatly obfuscates the function's intention. All this is avoided in Sisal, whose implementation is concise and easy to understand. The language's single-assignment nature does not preclude the new values from being stored back in PX. A smart run-time system can schedule

the concurrent array accesses to *PX* first, so that when the sums complete there are no longer any references to *old PX* and the array can be up-dated in place. The present compiler and run-time systems support such reference counting.

3.11 Loop 11

Loop 11 (Appendix A, page 37; Appendix C, page 66) returns the array *X* with *X(i)*, $i = \{2..n\}$, replaced by

$$\sum_{j=1}^{i} Y(i) \ .$$

There are a number of ways to express this function. The routine given in Figure 3a is maximally parallel, and is *O(Log n)* in time and $O(n^2)$ in number of operations. Figure 3b is the direct Fortran translation of the kernel. The routine is linear in both time and number of operations, and is likely to be less expensive than the former since it requires only a single processor. A third alternative is to use the method of recursive doubling. The Sisal code for this implementation is given in the appendix. Its structure is identical to that of Loop 5 and the reader is referred to Section 3.5 for an explanation. The function is *O(Log n)* in time and *O(n Log n)* in number of operations. Whether or not it is more efficient than the sequential routine will be system dependent.

Appendix C includes the execution trace of the recursive doubling version of Loop 11 *(n = 128)*. There are 7, or $\lceil Log\ n \rceil$, spikes. The width of each spike is 9 and the height of each is proportional to the parallelism of the inner loop on the corresponding iteration. Since each successive iteration calculates 2^i fewer values, the heights fall as a power of *2*. To complete execution in the length of the trace requires hundreds of processors; whereas, to complete execution of the sequential implementation [Figure 4] requires more time but only two processors. These graphs demonstrate clearly the trade-off between time and resources.

3.12 Loop 12

Loop 12 (Appendix A, page 37, Appendix C, page 67) returns a one-dimensional array of size *n*. The i^{th} element of the array is set to *Y[i + 1] - Y[i]*. A **for all** expression computes the values in parallel and gathers them into an array. The Sisal code is obviously *O(1)* and optimal.

```
% Loop 11 - First Sum (Parallel Version)
%
function Loop11p(n:integer; Y:OneDim; returns OneDim)

  for i in 1,n returns
  array of for j in 1,i returns
           value of sum Y[j]
           end for
  end for

end function %Loop11p
```

Figure 3a: Parallel Implementation of Loop 11

```
% Loop 11 - First Sum (Sequential Version)
%
function Loop11s(n:integer; Y:OneDim; returns OneDim)

  for initial
      i := 1;
      X := Y[1]
  while i < n repeat
      i := old i + 1;
      X := old X + Y[i]
  returns array of X
  end for

end function %Loop11s
```

Figure 3b: Sequential Implementation of Loop 11

3.13 Loop 13

Loop 13 (Appendix A, page 38; Appendix C, page 68) returns two two-dimensional arrays, $\underline{H}$ and $\underline{P}$. The function is sequential because $\underline{H}$ is indexed indirectly through $\underline{P}$. If it were known that every element of $\underline{H}$ was updated exactly once, the function could be parallelized. The Sisal code is $O(n)$ and optimal. Figure 5 gives the Fortran function MOD2N in Sisal. Despite the number and complexity of instructions there is very little parallelism in the loop; the statements follow one another in strict sequence. Even the replacement of elements in $\underline{old\ P}$ and $\underline{P1}$ is not done

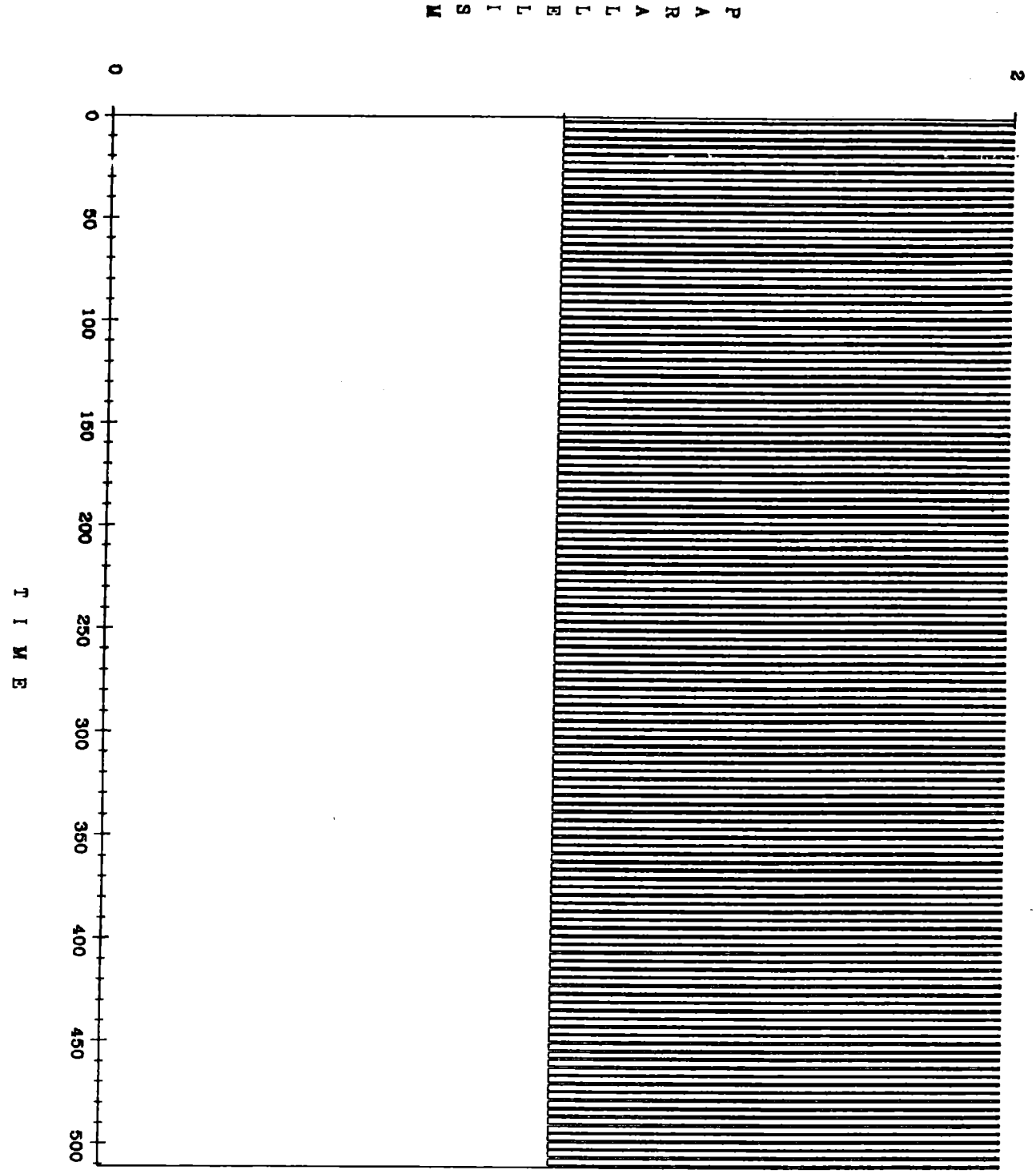


Figure 4: Loop 11 - First Sum (Sequential), n = 128

```
function MOD2N(i,j:integer; returns integer)

  if i < 0 then
     i - (i / j * j) + (j / 2) + abs(j/2)
  else
     i - (i / j * j) + (j / 2) - abs(j/2)
  end if

end function %MOD2N
```

Figure 5: The Sisal Function MOD2N

in parallel, but instead is a sequence of array replacements. In fact, to replace each element in *old P* requires 3 instructions: an **array access** to get the right row, an **array replace** to place the new value in the row, and a second **array replace** to place the row back into the matrix. Concurrent operations in the function are limited mostly to array accesses.

3.14 Loop 14

Loop 14 (Appendix A, page 38; Appendix C, page 69) returns nine one-dimensional arrays of size *n*. Eight of the arrays are loop independent and can be calculated in parallel despite stationary dependences between the array elements. The ninth computation updates *RH* indexed indirectly through *IR*. Since the elements of *IR* are not known a priori, we can not rule out multiple updates of the same element and must, therefore, execute the computation sequentially. If multiple updates could be ruled out, the computation of *RH* could be parallelized but would still have to execute after the others. Logically the function is *O(n)*.

In the Sisal implementation in Appendix A, a **for all** expression of arity eight computes the eight loop-independent arrays respecting the stationary dependencies between elements. The sequential loop in the **in** clause returns the new *RH* array. The Sisal function is *O(n)* and optimal. The execution trace in Appendix C is for $n = 1001$. The tall, thin spike is the **for all.** It has a width of 21 and a maximum height of 8008. The concurrency is due to the size of *n;* the stationary dependencies among the loop's statements restrict the parallelism per instance to a minimum. The long, flat tail depicts the sequential computation of *RH*. There are eight steps per element and a maximum of two concurrent operations per step. The uncertainty introduced by the indirect addressing of *RH* through *IR* destroys the function's parallelism. The kernel is an excellent

example of how badly the execution time of a parallel process can degrade when it contains even a single sequential computation of any magnitude.

3.15 Loop 15

Loop 15 (Appendix A, page 39; Appendix C, page 70) returns two two-dimensional arrays, *VS* and *VY*, of size *[2..n, 2..6]* and *[2..n, 2..7]*, respectively. The last row of *VS* and the last column of *VY* are zero filled. The Sisal implementation is *O(1)* and highly parallel. The original Fortran code is unnecessarily complicated; [3] gives a readable alternative. The Sisal version is simple, but its definition of arrays and lack of syntax makes appending a column of zeros to *VY* nontrivial. The two **for all** expressions in the **let** clause compute all but the last row of *VS* and last column of *VY*. We can append a row of zeros to *VS* with an **array_addh** operation since a two-dimensional array is an array of rows. But to append a column of zeros to *VY*, we must append a zero to each row. This requires that we scatter the rows, append a zero to each, and then regather the rows.

3.16 Loop 16

Loop 16 (Appendix A, page 40; Appendix C, page 71) searches for a particle in a two-dimensional gird divided into *ZONE[1]* zones. Each zone is subdivided into *n* groups. If the particle is found, the function returns the zone and group number of the location; else, it returns ZONE 1, GROUP 0. The Fortran code sequentially searches the groups and zones. The large number of arithmetic ifs make the code unreadable. The Sisal routine searches the groups and zones in parallel. Each instance of the search returns either the associated group number (the groups being numbered sequentially across the zones) or a number out of bounds. The results are then reduced to a single value by the **value of least** reduction operator. Finally, an **if** expression in the **let** clause tests the result: if it is out of bounds, 1 and 0 are returned; else, the appropriate zone and group number are returned.

Parallelizing the search simplifies the routine and brings out the nature of the zone-group partitioning of the search area. Moreover, the multiple-case **if** expression makes the decision logic of the search more understandable. The Sisal routine is *O(1)* and optimal. The execution trace in Appendix C is for two zones of 75 groups each. The graph is ragged because each loop body does not execute the same code. The graph is unusually wide because of the strictness of **if-then-else** expressions. The exact implementation of **if** expressions in Sisal is discussed in Section 3.22.

3.17 Loop 17

Loop 17 (Appendix A, page 40; Appendix C, page 72) returns three one-dimensional arrays of size *n: VE3, VXND,* and *VXNE*. The elements of the arrays are functions of *XNM,* whose value is recurrent; thus, we can not compute the elements in parallel. Moreover, we can not use recursive doubling because the boolean expression which controls how we compute the next elements of the arrays and the next value of *XNM* is a function of *XNM*. Since how the next values are computed depends on the previous value of *XNM,* no part of the computation can begin until that value is known; therefore, the iterations must execute in strict sequence.

The Sisal implementation is $O(n)$ and straightforward. The arrays returned by the function are in reverse order with respect to the arrays returned by the Fortran routine. We plan to include an option to gather array values in reverse order in the next release of the language. As Sisal is defined presently, the first value(s) returned by a **for** expression must appear in the initialization clause. This is inconvenient, a good source of errors, and not required by the compiler for any practical reason. We plan to drop the requirement in the next release and to allow the body of the loop to compute all return values. However, we will retain the option of including the first values in the initial clause, since often the first values of a recurrent computation are computed differently than the others, as in the sequential versions of Loops 5 and 11.

3.18 Loop 18

Loop 18 (Appendix A, page 41; Appendix C, page 73) returns six two-dimensional arrays of size *[1..n, 2..7]*. It is a course-grain pipeline procedure consisting of three logical sections: the first computes *ZA* and *ZB,* the second uses the results of the first to compute *ZU* and *ZV,* and the third uses the results of the second to compute *ZR* and *ZZ*. Although the three sections must execute sequentially, the computations within each section are data independent and can execute in parallel. In Fortran there is an anti-dependency between the first and third stages that forces the latter to execute nine steps behind the former. Since the second stage must trail the first stage by only two steps, a parallel Fortran implementation has at best three disjoint stages. But in a single-assignment language there are no anti-dependencies. Thus in Sisal, the third stage has only a stationary dependence with the second and so, may be merged with it.

Normally we would use streams in Sisal to implement a pipeline procedure, but because the $(i, j)^{th}$ calculation of the second stage uses nonconsecutive results of the first stage, a stream implementation is involved. A more straightforward procedure is given in the appendix. It is $O(1)$ and optimal. The function consists of two **for all** expressions of arity two and four, respectively.

The loops execute sequentially because of the data dependencies between their respective computations. The procedure's logical structure is implicit rather than explicit, as it would be in a stream implementation. The difficulty of replacing unnatural components of an array in Sisal complicates the routine. Since all elements of *ZA* and *ZB* are required in the second section, the first section must return the entire arrays and not just the new values. **SAReplace** performs the dissection and concatenation required to piece together the new arrays. The routine takes 14 steps to execute (steps 14 through 27, inclusive) independent of the size of the input parameters. If Sisal had syntax to express the replacement, it could be completed in one or two steps. It is paramount that such syntax be added to the language.

The execution trace shows that the procedure executes in two disjoint phases. We see such quantized bursts of parallelism throughout the traces. They are an artifact of the interpreter's strict implementation of loops, which makes no results available until the entire loop completes. Such strictness does limit parallelism. The computations of the second stage could begin as soon as the operations of the first stage on which they depend finish, they need not wait until the entire first stage completes. Although the interpreter implements loops strictly and the semantics of the language implies it, strict implementation of loops is not an axiom of the language and need not be guaranteed by the run time system.

In the case of Loop 18, even an unsophisticated run-time system could relax the strictness of the loops and execute the function optimally; however, we are lucky that what looks best to the user is easily analyzed. This is not always the case. As discussed in section 3.23, the user may sometimes turn somersaults to achieve a less complex implementation of a routine. The resulting code is so involved that the system can no longer identify the dependency between individual operations of different stages and must wait for each stage to complete before continuing. On the other hand, if the user ran the logically more complex but structurally simpler code, the run-time system would realize the dependencies and would execute the function optimally. If the language does not insist parallel loops be strict and run-time systems do not implement them so, some way for the user to express this nonstrictness should be found, so that he can design algorithms appropriately.

3.19 Loop 19

Loop 19 (Appendix A, page 42; Appendix C, page 74) returns a one-dimensional array, *B5*, of size *n*. The array is computed twice: first, in a forward loop from 1 to n, and then in a reverse loop from n to 1. The Sisal and Fortran implementations are drastically different.

The two computations expressed in Fortran are, respectively,

```
      Do 191 k = 1,n
         B5(k) = SA(k) + STB5 * SB(k)
191      STB5  = B5(k) - STB5
```

and

```
      Do 192 i = 1,n
         k = n - i + 1
         B5(k) = SA(k) + STB5 * SB(k)
192      STB5  = B5(k) - STB5
```

Obviously, both loops are recurrent in *STB5;* but by promoting *STB5* to a vector, both can be solved by recursive doubling. In the first loop, the vector equation for ***STB5*** is

$$\text{STB5(i)} = \begin{cases} \text{STIn} & \text{i} = 0 \\ \text{SA(i)} + \text{STB5(i-1)} * [\text{SB(i)} - 1.0\text{d0}] & 1 \le \text{i} \le \text{n} \end{cases}$$

and in the second loop, it is

$$\text{STB5(i)} = \begin{cases} \text{STB5}_1\text{(n)} & \text{i} = \text{n+1} \\ \text{SA(i)} + \text{STB5(i+1)} * [\text{SB(i)} - 1.0\text{d0}] & 1 \le \text{i} \le \text{n} \end{cases}$$

where the subscript 1 refers to a value computed in the first loop. With *STB5* a vector, the equation for ***B5*** in the second loop becomes

$$\text{B5(i)} = \text{SA(i)} + \text{STB5}_2\text{(i+1)} * \text{SB(i)} .$$

This equation is loop independent and all n values can be computed in parallel. It is not necessary to compute the equation for ***B5*** in the first loop since the second loop recomputes the entire array and neither ***STB5*** equation is a function of $B5_1$.

Both ***STB5*** equations are recurrent, but both can be solved by recursive doubling. The structures of the computations are similar to that of Loop 5 and the reader is referred to Section 3.5 for details. Since only the n^{th} value of $STB5_1$ is needed to compute $STB5_2$, we need only to compute that value and not the entire $STB5_1$ vector. Therefore, we need not execute the entire recursive doubling computation graph of the first loop, but only traverse the subtree that calculates the n^{th} value (Figure 6). **Tree19** is a recursive function that executes a parallel breadth-first traversal of the subtree. Each invocation returns a partial value of $STB5_1(n)$ and a product of *SBs*. At each level there are three possibilities, the node is either: a leaf $(h = 0)$, has only one descendent ($i \le 2^{h-1}$; i.e., the computation finished at a lower level), or has two descendents (the **else** clause).

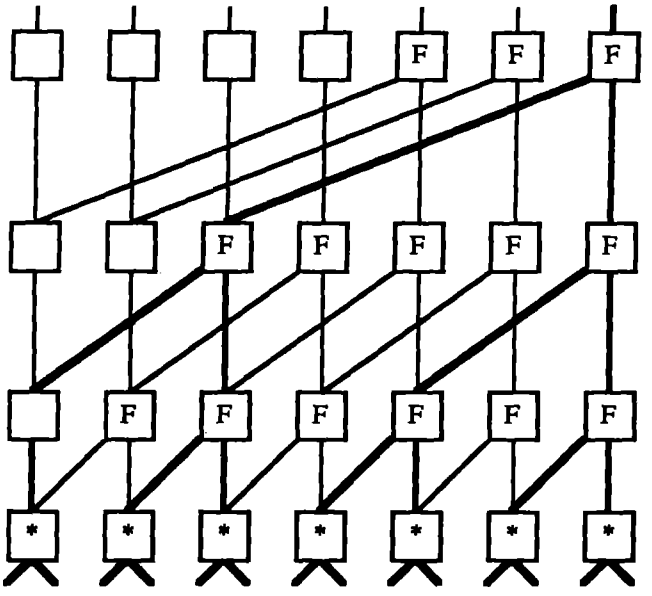


Figure 6: Computation Subtree for $STB5_1(n)$

On return of **Tree19,** the main function executes the computation graph of the second loop. We must compute the entire graph this time since all of $\underline{STB5_2}$ is needed to compute $\underline{B5}$. $\underline{STB5_2}$, is the concatenation of the array of n values returned by the **for** expression and the result of **Tree19.** The elements of $\underline{B5}$ are calculated simultaneously in a **for all** expression in the **in** clause of the **let** expression. Notice the bulk of the Sisal code computes $\underline{STB5}$; the calculation of *B5* is expressed in just two lines.

The Sisal implementation is $O(Log\ n)$ in time and $O(n\ Log\ n)$ in total number of operations; it is optimal. Despite the different components of the function, the code is readable and its structure is clear. The execution trace in Appendix C is for $n = 101$. It has three features, one for each of the function's three logical components. The first is the recursive traversal of the subtree to compute $STB5_1(n)$. Its height grows as we descend down the tree increasing the number nodes; and then shrinks as we climb back out. Its ragged shape is due to the unequal computations in the arms of the **if** expression. The rectangular feature at $t = 110$ is the **for all** expression in the initialization clause of the iterative loop. The next seven features depict the recursive doubling computation executed by the loop's body and are similar to the traces of Loop 5 and Loop 11. Notice 7 is $\lceil Log_2\ 101 \rceil$. The final feature is the parallel computation of $\underline{B5}$. Its shape is similar to the features

of the iterative loop because like the computation for *A*, it too consists of an addition and two array accesses, an array access, an addition, and a multiplication in that order.

3.20 Loop 20

Loop 20 (Appendix A, page 43; Appendix C, page 75) returns two one-dimensional arrays of size *n*: *X* and *XX*. The i^{th} value of *X* depends on the $(i - 1)^{st}$ value of *XX*, and the i^{th} value of *XX* depends on both the i^{th} value of *X* and the $(i - 1)^{st}$ value of *XX*. The loop-carried dependence in *XX* forces a sequential computation of the values. Recursive doubling can not be used in this case because the value of *DN* depends on a boolean expression that is a function of *XX*. Since the computation of *X* and *XX* need this value to start, no part of the i^{th} computation can begin until the $(i - 1)^{st}$ computation completes.

The Sisal routine is *O(n)* and easy to understand. The duplication of the loop's body in the initialization clause detracts from the code's appearance. As explained before, Sisal requires the initial clause of a **for** expression to compute the first return values. The function's trace shows there is little parallelism in evaluating the equations despite their complexity.

3.21 Loop 21

Loop 21 (Appendix A, page 43; Appendix C, page 76) computes the matrix equation,

$$PX - VY * CX.$$

The calculation is expressed entirely in the **returns** clause of the **for all** expression. It computes the inner products of the rows of *VY* and the columns of *CX* in parallel and then subtracts the results directly from the appropriate *PX* element. The optimizer moves the array accesses to *PX* and *VY* in the first dimension to the outer loop of the cross product of *i* and *j*. The Sisal function is *O(Log k)*, but since *k* is a constant *(= 14)*, the function is *O(1)*.

Originally, the interpreter ran for hours on this loop because arrays were constructed one element at a time. Consequently, the principle of single-assignment forced the interpreter to copy the result matrix each time it added a new element. With arrays now constructed as single objects, the computation completes in minutes.

3.22 Loop 22

Loop 22 (Appendix A, page 43; Appendix C, page 77) returns two one-dimensional arrays of size *n*: $\underline{Y}$ and $\underline{W}$. The appearance of $\underline{Y}$ in the equation for $\underline{W}$ creates a stationary dependency that forces the function to calculate the values of *Y(i)* and *W(i)* in tandem. The **if** expression in the Sisal routine is identical to the statement in the Fortran routine assuming the latter is sufficient to prevent division by 0; i.e., $V(i) = 0.0 \wedge U(i) < 0.0$ is never true. The Sisal function is $O(1)$ and optimal.

The loop's execution trace shows how DI implements **if** expressions and library function calls. The run is for $n = 101$. At time 0, 101 instances of the body are scattered. At step 1, each instance executes two array accesses, *V(i)* and *X(i)*, in parallel for a total of 202 operations. The array access *U(i)* is not executed at this time because instructions within **if** expressions are not hoisted. At steps 2 and 3, the interpreter evaluates the predicate and converts the boolean result to an integer, respectively, for a total of 101 operations per step. Since the predicate is always false, the **else** arm is always chosen. At step 4, the values of *U(i)* and *V(i)* are retrieved – a total of 202 operations. The array access *V(i)* is repeated since there is no common subexpression optimization across the boundaries of **if** expressions. The interpreter then computes the quotient of the two array elements and calls the library function **min.** Such functions appear in IF1 graphs as single nodes. Since we assume a uniform execution cost for all IF1 nodes, all library routines execute in a single time unit regardless of their complexity. Steps 7, 8, and 9, calculate the value of *W* and consist of 101 operations each. **exp** is another library function and it too executes in a single step. Finally at time 10, the interpreter gathers the results of the bodies into two arrays (202 operations); and at time 11, sets the lower bounds of the arrays to 1 (2 operations).

3.23 Loop 23

Loop 23 (Appendix A, page 44; Appendix C, page 78) is a wavefront algorithm. It consists of a series of parallel computations executing across the left-to-right diagonals of $\underline{ZA}$. Since the computations along each diagonal can occur in parallel, the logical complexity of the function is linear in the number of diagonal; that is $O(n + m)$, where *n* and *m* are the row and column order of $\underline{ZA}$, respectively [3].

The Sisal function in Appendix A returns $\underline{ZA}$ with the value of element (i, j), $2 \le i \le n - 1$, $2 \le j \le m - 1$, set to

```
ZA(i,j) + 0.175
*
[ZZ(i,j) - ZA(i,j) + ZA(i,j+1) * ZR(i,j)
+
ZA(i,j-1) * ZB(i,j) + ZA(i+1,j) * ZU(i,j) + ZA(i-1,j) * ZV(i,j)] .
```

The equation includes the element above, below, to the left, and to the right of *ZA(i, j);* namely, *ZA(k-1, j), ZA(k+1, j), ZA(k, j-1),* and *ZA(k, j+1),* respectively [Figure 7a]. Moreover, the values of *ZA(k-1, j)* and *ZA(k, j-1)* are new; and the values of *ZA(k+1, j)* and *ZA(k, j+1)* are old. This makes it impossible to compute either the rows or the columns of the new array in parallel; but, the computations along the left-to-right diagonals are data independent and can be computed in parallel [Figure 7b]. To insure that the elements lying above and to the left are new and that those lying below and to the right are old, the diagonals must be computed sequentially from the array's top left-hand corner to its bottom right-hand corner [Figure 7c].

The optimal Sisal implementation is an outer sequential loop stepping across the diagonals and an inner parallel loop computing the elements of the diagonals in parallel. Unfortunately, Sisal can not build two-dimensional arrays one diagonal at a time. The natural components of such arrays are elements and rows, not diagonals. A possible solution is to scatter out the entire array each iteration, compute the new values for the elements lying along a diagonal, and then recollect all the elements. This function completes in linear time but executes $O(n^3)$ operations. Because of the large number of operations, it is likely to be more expensive than a simple sequential version that computes the new values element-by-element. This direct Fortran translation (Figure 8) is quadratic in time and number of operations. However, the code's simplicity makes it probable that compile and run-time analysis could uncover the logical dependencies between iterations and achieve an optimal execution.

A better solution from the user's standpoint is to transform ZA into an array of diagonals, compute the new values, and then return the array to its original form. The Sisal code for this implementation is given in Appendix A. The procedure is now linear, but the code is complicated. A run-time system is unlikely to realize the dependencies between iterations and relax the strictness of the loops. Thus, the function may execute slower than the quadratic function discussed above.

The routine **Diagonals** (Figure 9a) takes a rectangular, two-dimensional array as input and returns a two-dimensional array whose rows are the left-to-right diagonals of the input array. **UnDiagonals** (Figure 9b) is the inverse function. Since the rows of DZA are the diagonals of ZA, the computation proceeds down the rows of DZA and replaces the elements in each row in

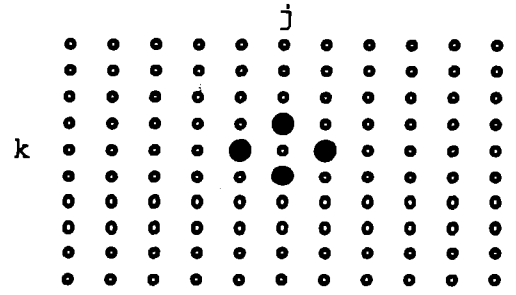


Figure 7a: Elements of ZA Required to Calculate ZA(k,j)

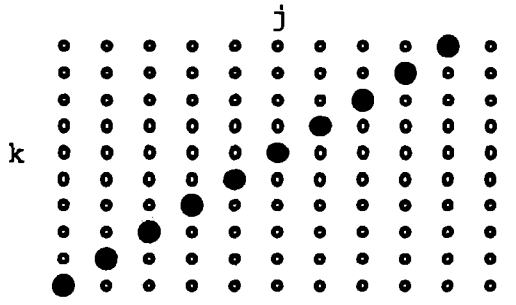


Figure 7b: A Left-to-Right Diagonal of ZA

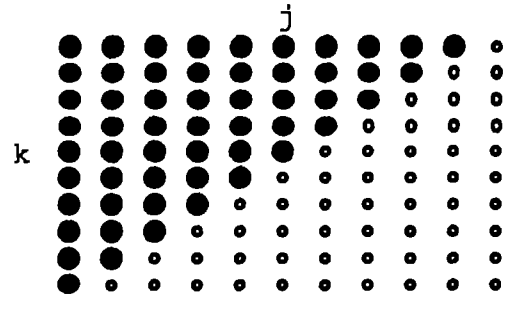


Figure 7c: Parallel Execution of Loop 23

```
% Loop 23 - 2-D Implicit Hydrodynamics (Sequential Version)
%
function Loop23s(n:integer; ZA,ZB:TwoDim;
                 ZR,ZU,ZV,ZZ:TwoDim; returns TwoDim)

  for initial
    i := 1
    R := ZA[1]
  while i < n repeat
    i := old i + 1;
    R := for initial
           j := 1;
           E := ZA[i,1]
         while j < 6 repeat
           j := old j + 1;
           E := ZA[i,j] + 0.175d0 * (ZZ[i,j] - ZA[i,j] +
                old E     * ZB[i,j] + old R[j]  * ZV[i,j] +
                ZA[i+1,j] * ZU[i,j] + ZA[i,j+1] * ZR[i,j])
         returns array of E
         end for
         || array[1: ZA[i,7]]
  returns array of R
  end for
  || array[1: ZA[n+1]]

end function %Loop23s
```

Figure 8: Sequential Implementation of Loop 23

---

parallel. The main function is *O(n)*. As written, both **Diagonals** and **UnDiagonals** are *O(1)* in time and $O(n^2)$ in number of operations. If we extend their execution times to *n*, we can reduce the number of instructions executed per time step to *O(n)*.

The execution trace of Loop 23 in Appendix C is for $n = 100$; ZA is dimensioned 101 by 7. The spikes at either end are the traces of **Diagonals** and **UnDiagonals,** respectively. Their n^2 nature is obvious. The graph between the spikes is the profile of the main loop. There are 106 features, one per diagonal. The height of each feature is directly proportional to the number of elements in the corresponding diagonal; they range from a relative height of 1 to 7. The width of each feature is constant. The discontinuity of the spikes is due to the strictness of the inner **for all** expression, which prevents any computation along the next diagonal from beginning until after all those along the preceding diagonal have completed. The graph shows the Sisal implementation is *O(n)*.

```
    for i in 1,r cross j in 1,min(i,c)
    returns array of A[i-j+1,j]
    end for
    ||
    for in in 2,c cross j in 1,c
    returns array of A[r+i-j,j]
    end for

end function %Diagonals
```

Figure 9a: The Sisal Function Diagonals

```
function UnDiagonals(r,c:integer;
                     A:TwoDim; returns TwoDim)

  for i in 1,r cross j in 1,c
  returns array of A[i+j-1,j]
  end for

end function %UnDiagonals
```

Figure 9b: The Sisal Function UnDiagonals

The need to "diagonalize" and "undiagonalize" ZA is due to the definition of arrays in Sisal. To build a composite object in parallel across anything other than its natural components is difficult. It is easy to build a two-dimensional array in parallel by computing either its elements or rows; but it is not easy to build the array in parallel across any other component, such as the diagonals or the quadrants. The problem is more than a deficiency in syntax as in the case of the array replace operations mentioned before. Even if one could syntactically access the diagonals, one could still not construct the new array one diagonal at a time. Instead, the values computed each iterations would have to be substituted for the old diagonal. Logically this can happen in place, but it is unlikely that even the most sophisticated analyzer would realize it. Thus, the run-time system would copy the entire array each iteration resulting in great inefficiencies.

Other data flow languages are not constrained in this manner. For example, the I-structures of Id Noveau [1,9] permit an arbitrary order of computation. The code for Loop 23 in Id Noveau is

```
Pnew = array(1..n, 1..m)
for i from 1 to n
for j from 1 to m
  if (i == 1) or (i == n) or
     (j == 1) or (j == m) then
       Pnew(i,j) = Pold(i,j)
  else Pnew(i,j) = Pold(i,j) + Pnew(i,j-1) +
                   Pnew(i-1,j) + Pold(i,j+1) + Pold(i+1,j)
return;
return;
```

The first statement creates an empty, two-dimensional matrix of size *n* by *m* in I-structure memory and the rest of the statements fill in the array. Notice the interior elements depend on elements of the new array; their computation is delayed until the elements on which they depend are available. There are no unnatural restrictions on how the computation progresses; the array is filled in naturally, constrained only by the existent data dependencies and the availability of values. In defense of Sisal, I-structures do allow possibly nondeterministic instruction sequences to be compiled.

3.24 Loop 24

Loop 24 (Appendix A, page 44; Appendix C, page 79) returns the first location of the minimum value of $\underline{X}$. The Sisal routine in the appendix is straightforward: the first loop finds the minimum value of $\underline{X}$; and the second loop returns the index of the first of those values. Both loops are parallel and use the **value of least** reduction operator to return the smallest value. The order of the loop is twice the order of the reduction operator, or *Log n*. The reduction anomaly renders the execution trace virtually meaningless.

There are many ways to write the loop in Sisal and Figure 10 gives two possible choices. The first is *O(n)*, and the second is *O(Log n)*. Which implementation is quickest on a particular system will depend on how efficiently that system executes each of the different constructs. The most elegant solution to Loop 24 is a single loop that returns the first index of the minimum value of $\underline{X}$. It is impossible to express this computation in Sisal. One can return the first minimum by comparing array elements from left-to-right (**returns value of left least ...**); but the reduction operator can return only the element's value and no other attribute. It can not return simultaneously both the element's value and its position in the array.

This function is important in data base applications. Often one wants to know the value of the i^{th} field of the record whose j^{th} field is minimum (or maximum). Sisal can find the appropriate record in parallel, but then must refind the record to return the i^{th} field. Note it is possible to return

```
% Loop 24 - Location of First Minimum (Sequential Version)
%
function Loop24s(n:integer; A:OneDim; returns integer)

  for initial
      i := 1;
      Loc := 1;
      Min := A[1]
  while i < n repeat
      i := old i + 1;
      Loc, Min := if A[i] < old Min then i, A[i]
                  else old Loc, old Min
                  end if
  returns value of Loc
  end for

end function %Loop24s
```

```
% Loop 24 - Location of First Minimum (Vector Version)
%
function Loop24v(n:integer; A:OneDim; returns integer)

  for initial
      nln := ln(real(n));
      size := exp(2, floor(nln(n)/ln(2.0) + 1);
      Min := A || array_fill(n + 1, size, A[1]);
      Index := for i in 1,size returns array of i end for
  while size > 1 repeat
      size := old size / 2;
      Min, Index :=
      for j in 1,size
          m, i := if old Min[2*j-1] <= old Min[2*j] then
                     old Min[2*j-1], old Index[2*j-1]
                  else
                     old Min[2*j], old Index[2*j]
                  end if
      returns array of m
              array of i
      end for
  returns value of Index[1]
  end for

end function %Loop24v
```

Figure 10: Alternate Implementations of Loop 24

the i^{th} field if the selection criteria is local (i.e., if it is not with respect to the entire data base) by appending the appropriate modifier to the expression in the **returns** clause. For example to return the first record whose j^{th} field is equal to 1, one would write

```
for i in 1,n
returns value of i when A[i,j] = 1
end for
```

4.0 Conclusions

This report presents the 24 Livermore Loops written in Sisal. We find the language flexible, easy to use, and able to express clearly and concisely the parallelism within the functions. All the Sisal implementations are of optimal order and straightforward; only in the case of Loop 23 did we have to work to achieve an optimal algorithm. In addition, Loops 2, 5, 6, 10, 11, 15, 16, 19, and 23 are nontrivial redesigns.

The report points out two major and a few minor deficiencies in the language. The major problems are: array syntax and the strictness of loops. Sisal limits array construction to elements, row, planes, and so on; the scientific user requires more flexible array access. Syntax must be developed (and it is being developed) to allows users to access matrices along arbitrary boundaries and construct matrices from components of arbitrary shape and size. From the user's point-of-view Sisal loops are strict, no result is available until the entire loop completes. Although current run-time systems relax the constraint, this is not sufficient. Since strictness is not a tenet of the language, some way for users to express nonstrict loops must be found. Otherwise, users will unnecessarily complicate their code attempting to circumvent the constraint.

The report demonstrates that Sisal is an effective language in which to express parallelism in scientific computations. However, one must redesign many algorithms with parallelism in mind starting with the problem's underlying mathematics or physics; one can not simply translate the procedures from a conventional language. We found such translations murky and nonoptimal. Parallel computing requires redesign of existing programs, not translation; compiler optimizations alone are insufficient. We can not stress this point enough.

Acknowledgements

I would like to thank the members of the Sisal group at the Laboratory; in particular, Kim Yates for making the necessary changes to DI for function traces, and to Steve Skedzielewski and Rod Oldehoeft for discussions concerning the particulars of Sisal and why things were designed the way they were. A special thanks to Frank McMahon for his comments and to Auda Motschenbacher for help in assembling the appendices.

References

1. Arvind, R.S. Nikhil, K.K. Pingali, "I-Structures: Data Structures for Parallel Computing", *Proceedings of the Workshop on Graph Reduction,* Los Alamos, New Mexico, September 1986.

2. Browne, J.C. "Framework for Formulation and Analysis of Parallel Computation Structures", *Parallel Computing,* Vol. 3, No. 1, January 1986, 1-9.

3. Feo, J.T., "An Analysis of the Computational and Parallel Complexity of the Livermore Loops", to appear *Parallel Computing,* 1987.

4. Gurd, J.R., C.C. Kirkham, and I. Watson, "The Manchester Prototype Dataflow Computer", *Communications of the ACM,* Vol. 28, January 1985, 34-52.

5. Kogge, H.S. and P.M. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations", *IEEE Transactions of Computers,* Vol. C-22, No. 8, August 1973, 786-793.

6. Kogge, H.S., "Parallel Solution of Recurrence Problems", *IBM Journal of Research Development,* March 1975, 138-148.

7. McGraw, J., et. al., *SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2,* M-146, Univ. of California - Lawrence Livermore National Laboratory, Livermore, California, March 1985.

8. McMahon, F.H., *Livermore FORTRAN Kernels: A Computer Test of the Numerical Performance Range,* UCRL-53745, University of California - Lawrence Livermore National Laboratory, Livermore, California, December 1986.

9. Nikhil, R.S., K.K. Pingali, and Arvind, *Id Nouveau,* CSG Memo 265, MIT Lab. for Computer Science, Cambridge, Massachusetts, July 1986.

10. Oldehoeft, R.R. and D.C. Cann, *Applicative Parallelism - Potential and Challenge,* UCRL - 96781, Univ. of California - Lawrence Livermore National Laboratory, Livermore, California, July 1987.

11. Skedzielewski, S.K. and J.R. Glauert, *IF1 - An Intermediate Form for Applicative Languages,* M-170, Univ. of California - Lawrence Livermore National Laboratory, Livermore, California, July 1985.

12. Skedzielewski, S.K., M.L. Welcome, "Dataflow Graph Optimization in IF1, *Functional Programming Languages and Computer Architecture,* ed. J. Jouannaud, Springer-Verlag, New York, New York, September 1985, 17-34.

13. Skedzielewski, S.K., R.K. Yates, and R.R. Oldehoeft, "DI: An Interactive Debugger Interpreter for Applicative Languages, *Proceedings of the ACM SIGPLAN 87 Symposium on Interpreters and Interpretive Techniques,* Minneapolis, Minnesota, June 1987, 102-109.

14. Welcome, M.L., et.al., *IF2: An Applicative Language Intermediate Form with Explicit Memory Management,* M-195, Univ. of California - Lawrence Livermore National Laboratory, Livermore, California, November 1986.

Appendix A: The Sisal Loops

```
% Loop 1 - Hydro Fragment                                              1
%                                                                      2
function Loop1(n:integer; Q,R,T:double;                                3
               Y,Z:OneDim; returns OneDim)                             4
  for k in 1,n returns                                                 5
  array of Q + (Y[k] * (R * Z[k+10] + T * Z[k+11]))                    6
  end for                                                              7
end function  %Loop1                                                   8
```

```
% Loop 2 - Incomplete Cholesky, Conjugate Gradient                     1
%                                                                      2
function Loop2(n:integer; V,X:OneDim; returns OneDim)                  3
  for initial                                                          4
      size := n;                                                       5
      index := 0;                                                      6
      nexts := array_adjust(X,1,n)                                     7
  while size > 1 repeat                                                8
      size  := old size / 2;                                           9
      index := old index + old size;                                  10
      next  := for j in 1,size - 1                                    11
               returns array of                                       12
                 old nexts[2 * j] -                                   13
                 V[old index + 2 * j] * old nexts[2 * j - 1] +        14
                 V[old index + 2 * j + 1] * old nexts[2 * j + 1]      15
               end for;                                               16
      j := 2 * size;                                                  17
      elem := if mod(old size, 2) then next[1]                        18
              else old nexts[j + 1]                                   19
              end if;                                                 20
      nexts := array_addh(next, old nexts[j] +                        21
                                V[old index + j + 1] * elem -         22
                                V[old index + j] * old nexts[j - 1])  23
  returns value of catenate nexts                                     24
  end for                                                             25
end function  %Loop2                                                  26
```

```
% Loop 3 - Inner Product                                               1
%                                                                      2
function Loop3(n: integer; X,Z: OneDim; returns double)                3
  for i in 1,n returns                                                 4
  value of sum X[i] * Z[i]                                             5
  end for                                                              6
end function  %Loop3                                                   7
```

```
% Loop 4 - Banded Linear Equations                                    1
%                                                                     2
function Loop4(n:integer; X,Y:OneDim; returns OneDim)                 3
  Let T6, T503, T1000 :=                                              4
         X[6] -    for i in 1,min(5, n/5)                             5
                   returns value of sum X[i] * Y[5*i]                 6
                   end for,                                           7
         X[503] -  for i in 1,min(5, n/5)                             8
                   returns value of sum X[497+i] * Y[5*i]             9
                   end for,                                          10
         X[1000] - for i in 1,min(5, n/5)                            11
                   returns value of sum X[994+i] * Y[5*i]            12
                   end for;                                          13
      S6, S503, S1000 :=                                             14
         if n/5 < 6 then                                             15
              T6, T503, T1000                                        17
         elseif n/5 = 6 then                                         18
              T6 - T6 * Y[30],                                       19
              T503 - T503 * Y[30],                                   20
              T1000 - T1000 * Y[30]                                  21
         else T6 - T6 * Y[30] - X[7] * Y[35],                        22
              T503 - T503 * Y[30] - X[504] * Y[35],                  23
              T1000 - T1000 * Y[30] - X[1001] * Y[35]                24
         end if                                                      25
  in  X[6: Y[5] * S6; 503: Y[5] * S503; 1000: Y[5] * S1000]          26
  end let                                                            27
end function  %Loop4                                                 28
```

```
% Loop 5 - Tri-Diangonal Elimination, Below Diagonal                 1
%                                                                     2
function Loop5(n:integer; X,Y,Z:OneDim; returns OneDim)               3
  for initial                                                         4
      i := 1;                                                         5
      Zs := array[1: 1.0d0] ||                                        6
            for i in 2,n returns array of -Z[i] end for;              7
      Xs := array[1: X[1]]  ||                                        8
            for i in 2,n returns array of Z[i] * Y[i] end for         9
  while i < n repeat                                                 10
      i := 2 * old i;                                                11
      Zs := array_adjust(old Zs, 1, old i) ||                        12
            for j in old i + 1, n returns                            13
            array of old Zs[j] * old Zs[j - old i]                   14
            end for;                                                 15
      Xs := array_adjust(old Xs, 1, old i) ||                        16
            for j in old i + 1, n returns                            17
            array of old Xs[j] + old Zs[j] * old Xs[j - old i]       18
            end for                                                  19
  returns value of Xs                                                20
  end for                                                            21
end function  %Loop5                                                 22
```

```
% Loop 6 - General Linear Recurrence Equations                          1
%                                                                       2
function Loop6(n:integer; B:TwoDim;                                     3
               WIn:OneDim; returns OneDim)                              4
  for initial                                                           5
      i := 1;                                                           6
      W := WIn                                                          7
  while i < n repeat                                                    8
      i := old i + 1;                                                   9
      W := for j in i,n returns                                        10
           array of old W[j] + B[j, j - old i] * old W[old i]           11
           end for                                                      12
  returns array of W[i]                                                13
  end for                                                              14
end function  %Loop6                                                   15
```

```
% Loop 7 - Equation of State Fragment                                   1
%                                                                       2
function Loop7(n:integer; R,T:double;                                   3
               U,Y,Z:OneDim; returns OneDim)                            4
  for k in 1,n returns                                                  5
  array of U[k] + R * (Z[k]   + R * Y[k])                               6
               + T * (U[k+3] + R * (U[k+2] + R * U[k+1])                7
               + T * (U[k+6] + R * (U[k+5] + R * U[k+4])))              8
  end for                                                               9
end function  %Loop7                                                   10
```

```
% Loop 8 - A. D. I. Integration                                         1
%                                                                       2
function Loop8(n:integer; A11,A12,A13,A21,A22,A23:double;               3
               A31,A32,A33,SIG:double; U1,U2,U3:ThreeDim;               4
               returns ThreeDim, ThreeDim, ThreeDim)                    5
  for kx in 2,3 cross ky in 2,n                                         6
      DU1 := U1[kx,ky+1,1] - U1[kx,ky-1,1];                             7
      DU2 := U2[kx,ky+1,1] - U2[kx,ky-1,1];                             8
      DU3 := U3[kx,ky+1,1] - U3[kx,ky-1,1]                              9
  returns                                                              10
      array of U1[kx,ky][2:                                            11
        U1[kx,ky,1] + A11 * DU1 + A12 * DU2 + A13 * DU3 +              12
        SIG * (U1[kx+1,ky,1] - 2.0d0 * U1[kx,ky,1] + U1[kx-1,ky,1])]   13
      array of U2[kx,ky][2:                                            14
        U2[kx,ky,1] + A21 * DU1 + A22 * DU2 + A23 * DU3 +              15
        SIG * (U2[kx+1,ky,1] - 2.0d0 * U2[kx,ky,1] + U2[kx-1,ky,1])]   16
      array of U3[kx,ky][2:                                            17
        U3[kx,ky,1] + A31 * DU1 + A32 * DU2 + A33 * DU3 +              18
        SIG * (U3[kx+1,ky,1] - 2.0d0 * U3[kx,ky,1] + U3[kx-1,ky,1])]   19
  end for                                                              20
end function  %Loop8                                                   21
```

```
% Loop 9 - Integrate Predictors                                          1
%                                                                        2
function Loop9(n:integer; CO,DM22,DM23,DM24,DM25:double;                 3
               DM26,DM27,DM28:double; PX:TwoDim; returns TwoDim)         4
  PX[1: for i in 1,n returns                                             5
        array of PX[3,i] +                                               6
                  CO    * PX[5,i]  + CO    * PX[6,i]  +                 7
                  DM22 * PX[7,i]  + DM23 * PX[8,i]  +                  8
                  DM24 * PX[9,i]  + DM25 * PX[10,i] +                  9
                  DM26 * PX[11,i] + DM27 * PX[12,i] +                  10
                  DM28 * PX[13,i]                                      11
        end for]                                                         12
end function  %Loop9                                                     13


% Loop 10 - Difference Predictors                                        1
%                                                                        2
function Loop10(n:integer; CX,PX:TwoDim; returns TwoDim)                 3
  for i in 5,14 cross j in 1,n returns                                   4
  array of CX[5,j] - for k in 6,i returns                                5
                     value of sum PX[k-1,j]                              6
                     end for                                             7
  end for                                                                8
end function  %Loop10                                                    9


% Loop 11 - First Sum                                                    1
%                                                                        2
function Loop11(n:integer; Y:OneDim; returns OneDim)                     3
  for initial                                                            4
      i := 1;                                                            5
      X := Y                                                             6
  while i < n repeat                                                     7
      i := 2 * old i;                                                    8
      X := array_adjust(old X, 1, old i) ||                              9
           for j in old i + 1,n returns                                  10
           array of old X[j] + old X[j - old i]                          11
           end for                                                       12
  returns value of X                                                     13
  end for                                                                14
end function  %Loop 11                                                   15


% Loop 12 - First Difference                                             1
%                                                                        2
function Loop12(n:integer; Y:OneDim; returns OneDim)                     3
  for i in 1,n returns                                                   4
  array of Y[i+1] - Y[i]                                                 5
  end for                                                                6
end function  %Loop 12                                                   7
```

```
% Loop 13 - Particle In Cell                                              1
%                                                                         2
function Loop13(n:integer;                                                3
                E,F:IntOneDim; B,C,HIn,PIn:TwoDim;                        4
                Y,Z:OneDim; returns TwoDim,TwoDim)                        5
  for initial                                                             6
      i := 0;                                                             7
      H := HIn;                                                           8
      P := PIn                                                            9
  while i < n repeat                                                      10
      i := old i + 1;                                                     11
      i1 := 1 + MOD2N(Trunc(old P[1,i]),64);                              12
      j1 := 1 + MOD2N(Trunc(old P[2,i]),64);                              13
      P1 := old P[4,i: old P[4,i] + C[i1,j1];                             14
                  3,i: old P[3,i] + B[i1,j1];                             15
                  2,i: old P[2,i] + old P[4,i] + C[i1,j1];                16
                  1,i: old P[1,i] + old P[3,i] + B[i1,j1]];               17
      i2 := MOD2N(Trunc(P1[1,i]),64);                                     18
      j2 := MOD2N(Trunc(P1[2,i]),64);                                     19
      i3 := i2 + E[i2+32];                                                20
      j3 := j2 + F[j2+32];                                                21
      P := P1[1,i: P1[1,i] + Y[i2+32];                                    22
             2,i: P1[2,i] + Z[j2+32]];                                    23
      H := old H[i3,j3: old H[i3,j3] + 1.0d0]                             24
  returns value of H                                                      25
          value of P                                                      26
  end for                                                                 27
end function  %Loop13                                                     28
```

```
% Loop 14 - Particle in Cell                                              1
%                                                                         2
function Loop14(n:integer; FLX:double;                                    3
                DEXIn,EXIn,GRD,RHIn:OneDim;                               4
                returns OneDim,OneDim,IntOneDim,IntOneDim,                5
                        OneDim,OneDim,OneDim,OneDim,OneDim)               6
  let DEX1,EX1,IR1,IX1,RX1,VX1,XI1,XX1 :=                                 7
      for i in 1,n                                                        8
          j := Trunc(GRD[i]);                                             9
          EX := EXIn[j];                                                  10
          DEX := DEXIn[j];                                                11
          XI := Double_Real(j);                                           12
          VX := EX - DEX * XI;                                            13
          k   := Trunc(VX + FLX);                                         14
          IR := MOD2N(k,512) + 1;                                         15
          RX := VX + FLX - Double_Real(k);                                16
          XX := VX + FLX - Double_Real(k) + Double_Real(IR)               17
```

```
      returns array of DEX                                                    18
              array of EX                                                     19
              array of IR                                                     20
              array of j                                                      21
              array of RX                                                     22
              array of VX                                                     23
              array of XI                                                     24
              array of XX                                                     25
      end for                                                                 26
  in  DEX1,EX1,IR1,IX1,                                                       27
      RX1 ,VX1,XI1,XX1,                                                       28
      for initial                                                             29
          i  := 0;                                                            30
          RH := RHIn                                                          31
      while i < n repeat                                                      32
          i  := old i + 1;                                                    33
          RH := old RH[IR1[i]:                                                34
                      old RH[IR1[i]] - RX1[i] + 1.0d0,                        35
                      old RH[IR1[i] + 1] + RX1[i]]                            36
      returns value of RH                                                     37
      end for                                                                 38
  end let                                                                     39
end function  %Loop14                                                         40
```

```
% Loop 15 - Casual Fortran. Development Version                               1
%                                                                             2
function Loop15(n:integer;                                                    3
                VF,VG,VH:TwoDim;                                              4
                returns TwoDim,TwoDim)                                        5
  let VS :=                                                                   6
      for i in 2,n-1 cross j in 2,6                                           7
        R,S,T := if VF[i,j] >= VF[i,j-1] then                                 8
                     max(VG[i,j],VG[i+1,j-1]),  VF[i,j],   0.053d0            9
                 else max(VG[i,j-1],VG[i+1,j-1]), VF[i,j-1], 0.073d0         10
                 endif                                                       11
      returns array of sqrt(VH[i,j] * VH[i,j] + R * R) * T / S               12
      end for;                                                                13
      VY :=                                                                   14
      for i in 2,n cross j in 2,6                                             15
        R,S := if VF[i,j] >= VF[i-1,j] then                                   16
                   Max(VH[i,j],VH[i,j+1]),     VF[i,j]                      17
               else Max(VH[i-1,j],VH[i-1,j+1]), VF[i-1,j]                     18
               end if;                                                        19
        T   := if VH[i,j+1] > VH[i,j] then 0.053d0 else 0.073d0 end if        20
      returns array of sqrt(VG[i,j] * VG[i,j] + R * R) * T / S                21
      end for                                                                 22
  in  array_addh(VS, array_fill(2, 6, 0.0d0)),                                23
      for row in VY returns array of array_addh(row, 0.0d0) end for           24
  end let                                                                     25
end function  %Loop15                                                         26
```

```
% Loop 16 - Monte Carlo Search Loop                                    1
%                                                                      2
function Loop16(n:integer; R,S,T:double; D,PLAN:OneDim;                3
                ZONE:IntOneDim; returns integer,integer)               4
  let Y := for i in 1,ZONE[1] cross j in 1,n                           5
           j4   := 2 * (n * (i - 1) + j - 1) + 3;                      6
           j5   := ZONE[2 * (n * (i - 1) + j - 1) + 3];                7
           test := D[j5] - (D[j5-1] * exp(T - D[j5-2], 2) +            8
                   exp(S - D[j5-3], 2) + exp(R - D[j5-4], 2));         9
           C1 := if j5 < n/3 then                                      10
                    if PLAN[j5] < T then ZONE[j4-1]                    11
                    elseif PLAN[j5] = T then 0                         12
                    else -ZONE[j4-1]                                   13
                    end if                                             14
                 elseif j5 < 2*n/3 then                                15
                    if PLAN[j5] < S then ZONE[j4-1]                    16
                    elseif PLAN[j5] = S then 0                         17
                    else -ZONE[j4-1]                                   18
                    end if                                             19
                 elseif j5 < n then                                    20
                    if PLAN[j5] < R then ZONE[j4-1]                    21
                    elseif PLAN[j5] = R then 0                         22
                    else -ZONE[j4-1]                                   23
                    end if                                             24
                 elseif j5 = n then 0                                  25
                 elseif test < 0.0d0 then ZONE[j4-1]                   26
                 else -ZONE[j4-1]                                      27
                 end if                                                28
         returns value of least if C1 = 0 then j4                      29
                                 else 2 * n * ZONE[1] + 2              30
                                 end if                                31
         end for                                                       32
  in  if Y = 2 * n * ZONE[1] + 2 then 1, 0                             33
      else (Y-3)/(2*n) + 1, Y                                          34
      end if                                                           35
  end let                                                              36
end function  %Loop16                                                  37


% Loop 17 - Implicit, Conditional Computation                          1
%                                                                      2
function Loop17(n:integer; VLIN,VLR,VSP,VSTP,VXNEIn:OneDim;            3
                returns OneDim, OneDim, OneDim)                        4
  for initial                                                          5
      i := n;                                                          6
      VXND := 1.03d0 / 3.07d0;                                         7
      E3 := VLR[n] / 3.0d0 + VLIN[n];                                  8
      E6 := VSP[n] / 3.0d0 + VSTP[n];                                  9
      XNM, VE3, VXNE :=                                                10
        if (0.2d0 > E3) |                                              11
           (0.6d0 * VXNEIn[n] > E3) then E6, E6, E6                    12
        else E3 + E3 - 0.33d0, E3, E3 + E3 - VXNEIn[n]                 13
        end if                                                         14
```

```
  while i > 2 repeat                                                    15
      i := old i - 1;                                                   16
      VXND := old XNM;                                                  17
      E3 := old XNM * VLR[i] + VLIN[i];                                 18
      E6 := old XNM * VSP[i] + VSTP[i];                                 19
      XNM, VE3, VXNE :=                                                 20
        if (0.6d0 * old XNM > E3) |                                     21
           (0.6d0 * VXNEIn[i] > E3) then E6, E6, E6                     22
        else E3 + E3 - old XNM, E3, E3 + E3 - VXNEIn[i]                 23
        end if                                                          24
  returns array of VE3                                                  25
          array of VXND                                                 26
          array of VXNE                                                 27
  end for                                                               28
end function  %Loop17                                                   29
```

```
% Loop 18 - 2-D Explicit Hydrodynamic Fragment                          1
%                                                                       2
function Loop18(n:integer; S,T:double;                                  3
                ZA,ZB,ZM,ZP,ZQ,ZR,ZU,ZV,ZZ:TwoDim;                      4
                returns TwoDim,TwoDim,TwoDim,TwoDim,TwoDim,TwoDim)      5
  let ZAcore, ZBcore :=                                                 6
      for i in 2,n cross j in 2,6 returns                               7
      array of (ZP[i-1,j+1] + ZQ[i-1,j+1] - ZP[i-1,j] - ZQ[i-1,j]) *    8
               (ZR[i,j] + ZR[i-1,j]) / (ZM[i-1,j] + ZM[i-1,j+1])        9
      array of (ZP[i-1,j] + ZQ[i-1,j] - ZP[i,j] - ZQ[i,j]) *           10
               (ZR[i,j] + ZR[i,j-1]) / (ZM[i,j] + ZM[i-1,j])            11
      end for;                                                          12
      ZANew, ZBNew := SAReplace(ZAcore,ZA), SAReplace(ZBcore,ZB)        13
  in  ZANew, ZBNew,                                                     14
      for i in 2,n cross j in 2,6                                       15
          ZUNew := ZU[i,j] + S * (                                      16
                   ZANew[i,j]   * (ZZ[i,j] - ZZ[i+1,j]) -               17
                   ZANew[i-1,j] * (ZZ[i,j] - ZZ[i-1,j]) -               18
                   ZBNew[i,j]   * (ZZ[i,j] - ZZ[i,j-1]) +               19
                   ZBNew[i,j+1] * (ZZ[i,j] - ZZ[i,j+1]));               20
          ZVNew := ZV[i,j] + S * (                                      21
                   ZANew[i,j]   * (ZR[i,j] - ZR[i+1,j]) -               22
                   ZANew[i-1,j] * (ZR[i,j] - ZR[i-1,j]) -               23
                   ZBNew[i,j]   * (ZR[i,j] - ZR[i,j-1]) +               24
                   ZBNew[i,j+1] * (ZR[i,j] - ZR[i,j+1]))                25
       returns array of ZUNew                                           26
               array of ZVNew                                           27
               array of ZR[i,j] + T * ZUNew                             28
               array of ZZ[i,j] + T * ZVNew                             29
      end for                                                           30
  end let                                                               31
end function  %Loop18                                                   32
```

```
% Loop 19 - General Linear Recurrence Equations                          1
%                                                                        2
function Tree19(i,h:integer; STIn:double;                                3
                SA,SB:OneDim; returns double, double)                    4
  if h = 0 then                                                          5
       if i > 1 then SA[i], SB[i] - 1.0d0                                6
       else SA[1] + STIn * (SB[1] - 1.0d0), 1.0d0                        7
       end if                                                            8
  elseif i <= exp(2, h-1) then                                           9
       Tree19(i,h-1,STIn,SA,SB)                                         10
  else let u,v := Tree19(i,h-1,STIn,SA,SB);                             11
           w,y := Tree19(i - exp(2,h-1),h-1,STIn,SA,SB)                 12
       in  u + v * w, v * y                                             13
       end let                                                          14
  end if                                                                15
end function %Tree19                                                    16



function Loop19(n:integer; STIn:double;                                  1
                SA,SB: OneDim; returns OneDim)                           2
  let h := trunc(ln(real(n))/ln(2.0)) + 1;                               3
      STB5n,T := Tree19(n, h, STIn, SA, SB);                             4
      STB5 := for initial                                                5
                i := 1;                                                  6
                A := SA[n: SA[n] + STB5n * (SB[n] - 1.0d0)];             7
                B := for j in i,n returns                                8
                     array of SB[j] - 1.0d0                              9
                     end for                                            10
                     || array[1: 1.0d0]                                 11
              while i < n repeat                                        12
                i := 2 * old i;                                         13
                j := n - old i;                                         14
                A := for k in 1,j returns                               15
                     array of old A[k] + old B[k] * old A[k + old i]    16
                     end for                                            17
                     || array_adjust(old A, j+1, n);                    18
                B := for k in 1,j returns                               19
                     array of old B[k] * old B[k + old i]               20
                     end for                                            21
                     || array_adjust(old B, j+1, n)                     22
              returns value of A                                        23
              end for                                                   24
              || array[1: STB5n]                                        25
  in  for i in 1,n returns                                              26
      array of SA[i] + STB5[i+1] * SB[i]                                27
      end for                                                           28
  end let                                                               29
end function  %Loop19                                                   30
```

```
% Loop 20 - Discrete Ordinates Transport: Conditional Recurrence on XX
%
function Loop20(n:integer; DK,S,T,XXIn:double;
                G,U,V,VX,W,Y,Z:OneDim; returns OneDim, OneDim)
  for initial
      i := 1;
      DI := Y[1] - G[1] / (XXIn + DK);
      DN := if DI = 0.0d0 then 0.20d0
            else max(S, min(Z[1]/DI, T))
            end if;
      X  := (XXIn * (W[1] + DN * V[1]) + U[1]) / (VX[1] + DN * V[1]);
      XX :=  XXIn + DN * (X - XXIn)
  while i < n repeat
      i := old i + 1;
      DI := Y[i] - G[i] / (old XX + DK);
      DN := if DI = 0.0d0 then 0.20d0
            else max(S, min(Z[i]/DI, T))
            end if;
      X  := (old XX * (W[i] + DN*V[i]) + U[i]) / (VX[i] + DN * V[i]);
      XX :=  old XX + DN * (X - old XX)
  returns array of X
          array of XX
  end for
end function  %Loop20
```

```
% Loop 21 - Matrix * Matrix Product
%
function Loop21(n:integer; CX,PX,VY:TwoDim; returns TwoDim)
  for i in 1,15 cross j in 1,n returns
  array of PX[i,j] + for k in 1,15 returns
                      value of sum VY[i,k] * CX[k,j]
                      end for
  end for
end function  %Loop21
```

```
% Loop 22 - Planckian Distribution
%
function Loop22(n:integer; U,V,X:OneDim; returns OneDim, OneDim)
  for i in 1,n
      Y := if V[i] = 0.0d0 then 20.0d0
           else min(U[i]/V[i], 20.0d0)
           end if;
      W := X[i] / (exp(2.71828182845905d0,Y) - 1.0d0)
  returns array of W
          array of Y
  end for
end function  %Loop22
```

```
% Loop 23 - 2-D Implicit Hydrodynamics                                    1
%                                                                         2
function Loop23(n:integer; ZA,ZB:TwoDim;                                  3
                ZR,ZU,ZV,ZZ:TwoDim; returns TwoDim)                       4
  let r := array_limh(ZA);                                                5
      c := array_limh(ZA[1]);                                             6
      DZA := Diagonals(r,c,ZA);                                           7
      NZA :=                                                              8
      for initial                                                         9
        i := 1;                                                           10
        Row := DZA[1]                                                     11
      while i < r + c - 1 repeat                                          12
        i := old i + 1;                                                   13
        l := array_liml(DZA[i]);                                          14
        h := array_limh(DZA[i]);                                          15
        Row := for j in l,h                                               16
                 k := i - j + 1                                           17
               returns array of                                           18
                 if (j = l) | (j = h) then DZA[i,j]                       19
                 else DZA[i,j] + 0.175d0 * (ZZ[k,j] - DZA[i,j] +          20
                      old Row[j] * ZV[k,j] + old Row[j-1] * ZB[k,j] +     21
                      DZA[i+1,j] * ZU[k,j] + DZA[i+1,j+1] * ZR[k,j])      22
                 end if                                                   23
               end for                                                    24
      returns array of Row                                               25
      end for                                                            26
  in  UnDiagonals(r,c,NZA)                                               27
  end let                                                                28
end function  %Loop23                                                    29
```

```
% Loop 24 - Location of First Minimum                                     1
%                                                                         2
function Loop24(n:integer; A:OneDim; returns integer)                     3
  let x := for y in A returns                                             4
           value of least y                                               5
           end for                                                        6
  in  for y in A at i returns                                             7
      value of least i when y = x                                         8
      end for                                                             9
  end let                                                                 10
end function  %Loop24                                                     11
```

Appendix B: The Fortran Loops

```
      Subroutine Loop1(n,Q,R,T,X,Y,Z)                                    1
      integer n,k                                                        2
      double precision Q,R,T,X(1001),Y(1001),Z(1001)                     3
      Do 1 k = 1,n                                                       4
1        X(k) = Q + (Y(k) * ((R * Z(k+10)) + (T * Z(k+11))))             5
      return                                                             6
      end                                                                7


      subroutine Loop2(n,V,X)                                            1
      integer n,k,i,IL,IPNT,IPNTP                                        2
      double precision V(1001),X(1001)                                   3
      IL = n                                                             4
      IPNTP = 0                                                          5
21    IPNT = IPNTP                                                       6
      IPNTP = IPNTP + IL                                                 7
      IL = IL / 2                                                        8
      i = IPNTP                                                          9
      Do 20 k = IPNT+2,IPNTP,2                                          10
         i = i + 1                                                      11
20       X(i) = X(k) - (V(k) * X(k-1)) + (V(k+1) * X(k+1))              12
      If(IL .GT. 1)  GOTO 21                                            13
      return                                                            14
      end                                                               15


      subroutine Loop3(n,Q,X,Z)                                          1
      integer n                                                          2
      double precision Q,X(1001),Z(1001)                                 3
      Q = 0.0d0                                                          4
      Do 3 k = 1,n                                                       5
3        Q = Q + (Z(k) * X(k))                                           6
      return                                                             7
      end                                                                8


      subroutine Loop4(n,X,Y)                                            1
      integer n,k,j,lw                                                   2
      double precision X(1001),Y(1001)                                   3
      Do 40 k = 7,1001,(1001 - 7)/2                                      4
         lw = k - 6                                                      5
         Do 41 j = 5,n,5                                                 6
            X(k-1) = X(k-1) - (X(lw) * Y(j))                             7
41          lw = lw + 1                                                  8
40       X(k-1) = Y(5) * X(k-1)                                          9
      return                                                            10
      end                                                               11
```

```
      subroutine Loop5(n,X,Y,Z)                                          1
      integer i,n                                                        2
      double precision X(1001),Y(1001),Z(1001)                           3
      Do 5 i = 2,n                                                       4
5        X(i) = Z(i) * (Y(i) - X(i-1))                                   5
      return                                                             6
      end                                                                7

      subroutine Loop6(n,B,W)                                            1
      integer n                                                          2
      double precision B(64,64),W(1001)                                  3
      Do 6 i = 2,n                                                       4
      Do 6 k = 1,i-1                                                     5
6        W(i) = W(i) + (B(i,k) * W(i-k))                                 6
      return                                                             7
      end                                                                8

      subroutine Loop7(n,R,T,U,X,Y,Z)                                    1
      integer n                                                          2
      double precision R,T,U(1001),X(1001),Y(1001),Z(1001)               3
      Do 7 k = 1,n                                                       4
         X(k) = U(k) +                                                   5
     .      (R * (Z(k) + (R * Y(k)))) +                                  6
     .      (T * (U(k+3) + (R * (U(k+2) + (R * U(k+1))))) +              7
     .           (T * (U(k+6) + (R * (U(k+5) + (R * U(k+4))))))))       8
7        Continue                                                        9
      return                                                            10
      end                                                               11

      subroutine Loop8(n,A11,A12,A13,A21,A22,A23,                        1
     .                 A31,A32,A33,SIG,U1,U2,U3)                         2
      integer n,kx,ky,nl1,nl2                                            3
      double precision DU1(101),DU2(101),DU3(101)                        4
      double precision U1(5,101,2),U2(5,101,2),U3(5,101,2)               5
      double precision A11,A12,A13,A21,A22,A23,A31,A32,A33,SIG           6
      nl1 = 1                                                            7
      nl2 = 2                                                            8
      Do 8 kx = 2,3                                                      9
      Do 8 ky = 2,n                                                     10
         DU1(ky) = U1(kx,ky+1,nl1) - U1(kx,ky-1,nl1)                    11
         DU2(ky) = U2(kx,ky+1,nl1) - U2(kx,ky-1,nl1)                    12
         DU3(ky) = U3(kx,ky+1,nl1) - U3(kx,ky-1,nl1)                    13
         U1(kx,ky,nl2) = U1(kx,ky,nl1) + (A11 * DU1(ky)) + (A12 *       14
     .     DU2(ky)) + (A13 * DU3(ky)) + (SIG * (U1(kx+1,ky,nl1) -       15
     .     (2.0d0 * U1(kx,ky,nl1)) + U1(kx-1,ky,nl1)))                  16
         U2(kx,ky,nl2) = U2(kx,ky,nl1) + (A21 * DU1(ky)) + (A22 *       17
     .     DU2(ky)) + (A23 * DU3(ky)) + (SIG * (U2(kx+1,ky,nl1) -       18
     .     (2.0d0 * U2(kx,ky,nl1)) + U2(kx-1,ky,nl1)))                  19
         U3(kx,ky,nl2) = U3(kx,ky,nl1) + (A31 * DU1(ky)) + (A32 *       20
     .     DU2(ky)) + (A33 * DU3(ky)) + (SIG * (U3(kx+1,ky,nl1) -       21
     .     (2.0d0 * U3(kx,ky,nl1)) + U3(kx-1,ky,nl1)))                  22
8        Continue                                                       23
      return                                                            24
      end                                                               25
```

```
      subroutine Loop9(n,CO,DM22,DM23,DM24,DM25,DM26,DM27,DM28,PX)
      integer i,n
      double precision CO,DM22,DM23,DM24,DM25,DM26,DM27,DM28,PX(25,101)
      Do 9 i = 1,n
         PX(1,i) = PX(3,i) +  (CO * (PX(5,i) + PX(6,i))) +
     .             (DM28 * PX(13,i)) + (DM27 * PX(12,i)) +
     .             (DM26 * PX(11,i)) + (DM25 * PX(10,i)) +
     .             (DM24 * PX(9,i))  + (DM23 * PX(8,i))  +
     .             (DM22 * PX(7,i))
9        Continue
      return
      end
```

```
      subroutine Loop10(n,CX,PX)
      integer i,n
      double precision AR,BR,CR,CX(25,101),PX(25,101)
      Do 10 i = 1,n
         AR = CX(5,i)
         BR = AR - PX(5,i)
         PX(5,i) = AR
         CR = BR - PX(6,i)
         PX(6,i) = BR
         AR = CR - PX(7,i)
         PX(7,i) = CR
         BR = AR - PX(8,i)
         PX(8,i) = AR
         CR = BR - PX(9,i)
         PX(9,i) = BR
         AR = CR - PX(10,i)
         PX(10,i) = CR
         BR = AR - PX(11,i)
         PX(11,i) = AR
         CR = BR - PX(12,i)
         PX(12,i) = BR
         PX(14,i) = CR - PX(13,i)
10       PX(13,i) = CR
      return
      end
```

```
      subroutine Loop11(n,X,Y)
      integer k,n
      double precision X(1001),Y(1001)
      X(1) = Y(1)
      Do 11 k = 2,n
11       X(k) = X(k-1) + Y(k)
      return
      end
```

```
      subroutine Loop12(n,X,Y)                                          1
      integer k,n                                                       2
      double precision X(1001),Y(1001)                                  3
      Do 12 k = 1,n                                                     4
12       X(k) = Y(k+1) - Y(k)                                           5
      return                                                            6
      end                                                               7
```

```
      subroutine Loop13(n,E,F,B,C,H,P,Y,Z)                              1
      integer n,ip,i1,j1,i2,j2,E(96),F(96)                              2
      double precision B(64,64),C(64,64),H(64,64)                       3
      double precision P(4,512),Y(1001),Z(1001)                         4
      Do 13 ip = 1,n                                                    5
         i1 = P(1,ip)                                                   6
         j1 = P(2,ip)                                                   7
         i1 = 1 + MOD2N(i1,64)                                          8
         j1 = 1 + MOD2N(j1,64)                                          9
         P(3,ip) = P(3,ip) + B(i1,j1)                                  10
         P(4,ip) = P(4,ip) + C(i1,j1)                                  11
         P(1,ip) = P(1,ip) + P(3,ip)                                   12
         P(2,ip) = P(2,ip) + P(4,ip)                                   13
         i2 = P(1,ip)                                                  14
         j2 = P(2,ip)                                                  15
         i2 = MOD2N(i2,64)                                             16
         j2 = MOD2N(j2,64)                                             17
         P(1,ip) = P(1,ip) + Y(i2+32)                                  18
         P(2,ip) = P(2,ip) + Z(j2+32)                                  19
         i2 = i2 + E(i2+32)                                            20
         j2 = j2 + F(j2+32)                                            21
13       H(i2,j2) = H(i2,j2) + 1.0d0                                   22
      return                                                           23
      end                                                              24
```

```
      subroutine Loop14(n,FLX,DEX,DEX1,EX,EX1,                          1
                        GRD,IR,IX,RH,RX,VX,XI,XX)                       2
      integer n,k,IR(1001),IX(1001)                                     3
      double precision FLX,DEX(1001),DEX1(1001)                         4
      double precision EX(1001),EX1(1001),GRD(1001)                     5
      double precision RH(1001),RX(1001),VX(1001),XI(1001),XX(1001)     6
      Do 141 k = 1,n                                                    7
         VX(k) = 0.0d0                                                  8
         XX(k) = 0.0d0                                                  9
         IX(k) = INT(GRD(k))                                           10
         XI(k) = FLOAT(IX(k))                                          11
         EX1(k) = EX(IX(k))                                            12
141      DEX1(k) = DEX(IX(k))                                          13
```

```
      Do 142 k = 1,n                                                    14
         VX(k) = VX(k) + EX1(k) + (DEX1(k) * (XX(k) - XI(k)))           15
         XX(k) = XX(k) + VX(k) + FLX                                    16
         IR(k) = XX(k)                                                  17
         RX(k) = XX(k) - IR(k)                                          18
         IR(k) = MOD2N(IR(k),512) + 1                                   19
142      XX(k) = RX(k) + IR(k)                                          20
      Do 140 k = 1,n                                                    21
         RH(IR(k)) = RH(IR(k)) - RX(k) + 1.0d0                          22
140      RH(IR(k) + 1) = RH(IR(k) + 1) + RX(k)                          23
      return                                                            24
      end                                                               25
```

```
      subroutine Loop15(n,VF,VG,VH,VS,VY)                               1
      integer j,k                                                       2
      double precision R,S,T,VF(101,7),VG(101,7)                        3
      double precision VH(101,7),VS(101,7),VY(101,25)                   4
      do 1500 j = 2,7                                                   5
      do 1500 k = 2,n                                                   6
         if(j-7) 1502,1501,1501                                         7
1501     VY(k,j) = 0.0d0                                                8
         goto 1500                                                      9
1502     if(VH(k,j+1) - VH(k,j)) 1504,1504,1503                         10
1503     T = 0.053d0                                                    11
         goto 1505                                                      12
1504     T = 0.073d0                                                    13
1505     if(VF(k,j) - VF(k-1,j)) 1506,1507,1507                         14
1506     R = max(VH(k-1,j), VH(k-1,j+1))                                15
         S = VF(k-1,j)                                                  16
         goto 1508                                                      17
1507     R = max(VH(k,j), VH(k,j+1))                                    18
         S = VF(k,j)                                                    19
1508     VY(k,j) = dsqrt(VG(k,j)**2 + R * R) * T / S                    20
1509     if(k-n) 1511,1510,1510                                         21
1510     VS(k,j) = 0.0d0                                                22
         goto 1500                                                      23
1511     if(VF(k,j) - VF(k,j-1)) 1512,1513,1513                         24
1512     R = max(VG(k,j-1), VG(k+1,j-1))                                25
         S = VF(k,j-1)                                                  26
         T = 0.073d0                                                    27
         goto 1514                                                      28
1513     R = max(VG(k,j), VG(k+1,j))                                    29
         S = VF(k,j)                                                    30
         T = 0.053d0                                                    31
1514     VS(k,j) = dsqrt(VH(k,j)**2 + R * R) * T / S                    32
1500  continue                                                          33
      return                                                            34
      end                                                               35
```

```
      subroutine Loop16(n,m,j4,R,S,T,D,PLAN,ZONE)                          1
      integer n,m,i1,j2,j4,j5,k,II,LB,ZONE(300)                            2
      double precision R,S,T,D(300),PLAN(300)                              3
      m = 1                                                                4
      i1 = m                                                               5
      II = n / 3                                                           6
      LB = II + II                                                         7
1601  j2 = (n + n) * (m - 1) + 1                                           8
      do 1613 k = 1,n                                                      9
         j4 = j2 + k + k                                                  10
         j5 = ZONE(j4)                                                    11
         if(j5 - n) 1603,1614,1609                                        12
1602     if(j5 - n + II) 1605,1604,1604                                   13
1603     if(j5 - n + LB) 1606,1602,1602                                   14
1604     if(PLAN(j5) - R) 1608,1615,1607                                  15
1605     if(PLAN(j5) - S) 1608,1615,1607                                  16
1606     if(PLAN(j5) - T) 1608,1615,1607                                  17
1607     if(ZONE(j4 - 1)) 1610,1616,1613                                  18
1608     if(ZONE(j4 - 1)) 1613,1616,1610                                  19
1609     if(D(j5) - (D(j5-1) * (T - D(j5-2))**2 +                         20
     .     (S - D(j5-3))**2 + (R - D(j5-4))**2)) 1608,1615,1607           21
1610     m = m + 1                                                        22
         if(m - ZONE(1)) 1612,1612,1611                                   23
1611     m = 1                                                            24
1612     if(i1 - m) 1601,1615,1601                                        25
1613  continue                                                            26
1614  continue                                                            27
1615  continue                                                            28
1616  continue                                                            29
      return                                                              30
      end                                                                 31
```

```
      subroutine Loop17(n,VE3,VLIN,VLR,VSP,VSTP,VXND,VXNE)                1
      integer i,n                                                          2
      double precision E3,E6,XNC,XNEI,XNM,VE3(101),VLIN(101)               3
      double precision VLR(101),VSP(101),VSTP(101),VXND(101),VXNE(101)     4
      i = n                                                                5
      XNM = 1.0d0 / 3.0d0                                                  6
      E6 = 1.03d0 / 3.07d0                                                 7
      goto 172                                                             8
171   E6 = XNM * VSP(i) + VSTP(i)                                          9
      VXNE(i) = E6                                                        10
      XNM = E6                                                            11
      VE3(i) = E6                                                         12
      i = i - 1                                                           13
      if(i .eq. 1) goto 170                                               14
```

```
172  E3 = XNM * VLR(i) + VLIN(i)                                          15
     XNC = 5.0d0 / 3.0d0 * E3                                             16
     XNEI = VXNE(i)                                                       17
     VXND(i) = E6                                                         18
     if(XNM .gt. XNC) goto 171                                            19
     if (XNEI .gt. XNC) goto 171                                          20
     VE3(i) = E3                                                          21
     E6 = E3 + E3 - XNM                                                   22
     VXNE(i) = E3 + E3 - XNEI                                             23
     XNM = E6                                                             24
     i = i - 1                                                            25
     if(i .ne. 1) goto 172                                                26
170  continue                                                             27
     return                                                               28
     end                                                                  29
```

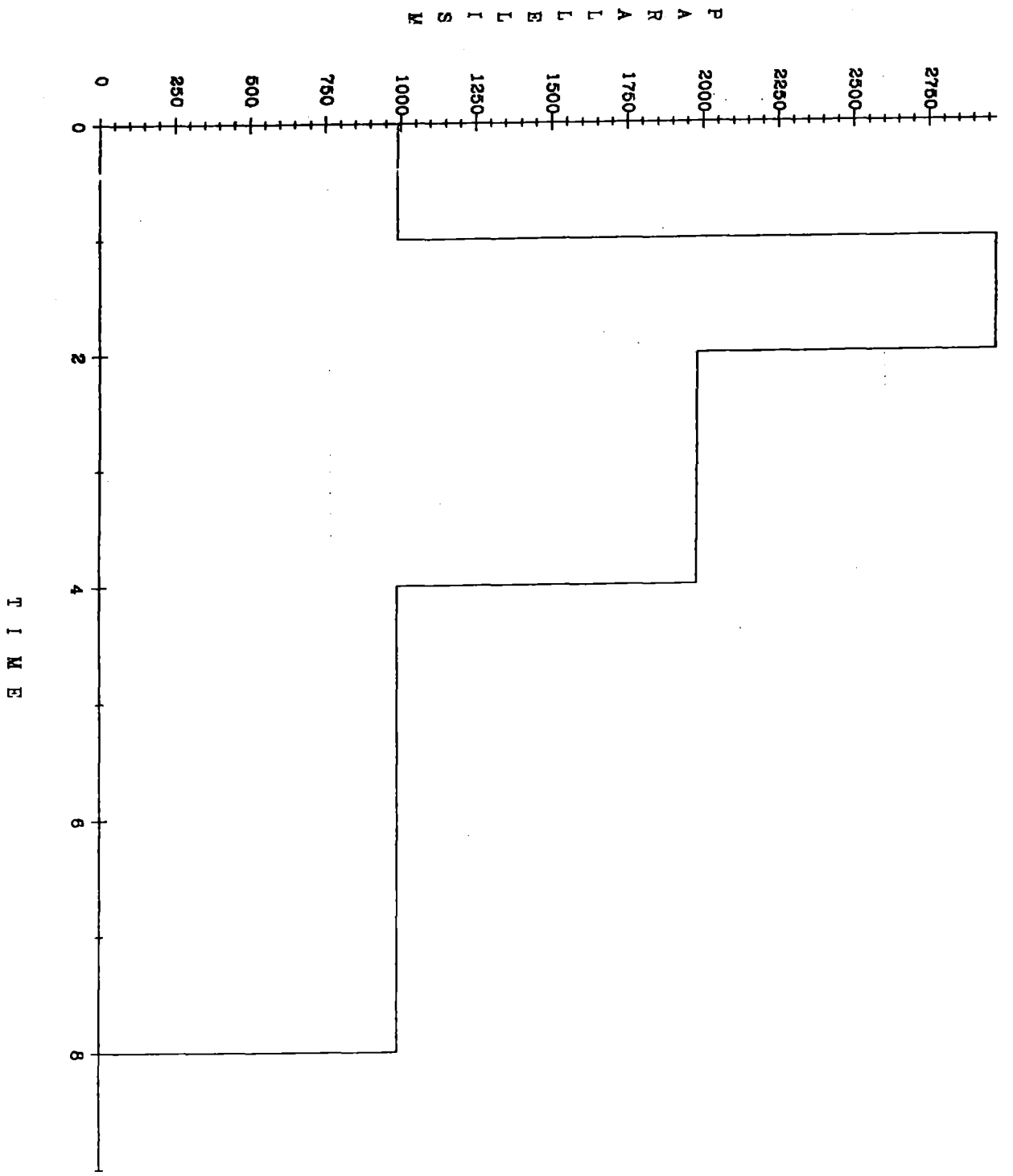
```
     subroutine Loop18(n,S,T,ZA,ZB,ZM,ZP,ZQ,ZR,ZU,ZV,ZZ)                  1
     integer k,j,n                                                        2
     double precision S,T,ZA(101,7),ZB(101,7),ZM(101,7),ZP(101,7)         3
     double precision ZQ(101,7),ZR(101,7),ZU(101,7),ZV(101,7),ZZ(101,7)   4
     do 181 k = 2,6                                                       5
     do 181 j = 2,n                                                       6
        ZA(j,k) =                                                         7
    .   (ZP(j-1,k+1) + ZQ(j-1,k+1) - ZP(j-1,k) - ZQ(j-1,k)) *             8
    .   (ZR(j,k) + ZR(j-1,k)) / (ZM(j-1,k) + ZM(j-1,k+1))                 9
        ZB(j,k) =                                                        10
    .   (ZP(j-1,k) + ZQ(j-1,k) - ZP(j,k) - ZQ(j,k)) *                    11
    .   (ZR(j,k) + ZR(j,k-1)) / (ZM(j,k) + ZM(j-1,k))                    12
181  continue                                                            13
     do 182 k = 2,6                                                      14
     do 182 j = 2,n                                                      15
        ZU(j,k) = ZU(j,k) + S *                                          16
    .             (ZA(j,k)   * (ZZ(j,k) - ZZ(j+1,k)) -                   17
    .              ZA(j-1,k) * (ZZ(j,k) - ZZ(j-1,k)) -                   18
    .              ZB(j,k)   * (ZZ(j,k) - ZZ(j,k-1)) +                   19
    .              ZB(j,k+1) * (ZZ(j,k) - ZZ(j,k+1)))                    20
```

```
      subroutine Loop19(n,STB5,B5,SA,SB)                                 1
      integer i,k,n                                                      2
      double precision STB5,B5(101),SA(101),SB(101)                      3
      do 191 k = 1,n                                                     4
         B5(k) = SA(k) + STB5 * SB(k)                                    5
191      STB5 = B5(k) - STB5                                             6
      do 193 i = 1,n                                                     7
         k = n - i + 1                                                   8
         B5(k) = SA(k) + STB5 * SB(k)                                    9
193      STB5 = B5(k) - STB5                                            10
      return                                                             11
      end                                                                12


      subroutine Loop20(n,DK,S,T,G,U,V,VX,W,X,XX,Y,Z)                    1
      integer k,n                                                        2
      double precision DK,DI,DN,S,T,G(1001),U(1001),V(1001),VX(1001)     3
      double precision W(1001),X(1001),XX(1001),Y(1001),Z(1001)          4
      do 20 k = 1,n                                                      5
         DI = Y(k) - (G(k) / (XX(k) + DK))                               6
         DN = 0.2d0                                                      7
         if(DI .NE. 0.0d0) DN = max(s, min(Z(k)/DI, T))                  8
         X(k) = ((W(k) + V(k) * DN) * XX(k) + U(k)) /                    9
                (VX(k) + V(k) * DN)                                     10
         XX(k+1) = (X(k) - XX(k)) * DN + XX(k)                          11
20     continue                                                         12
       return                                                           13
       end                                                              14


      subroutine Loop21(n,CX,PX,VY)                                      1
      integer i,j,k,n                                                    2
      double precision CX(25,101),PX(25,101),VY(101,25)                  3
      do 21 k = 1,25                                                     4
      do 21 i = 1,25                                                     5
      do 21 j = 1,n                                                      6
         PX(i,j) = PX(i,j) + VY(i,k) * CX(k,j)                           7
21    continue                                                           8
      return                                                             9
      end                                                               10


      subroutine Loop22(n,U,V,W,X,Y)                                     1
      integer k,n                                                        2
      double precision U(1001),V(1001),W(1001),X(1001),Y(1001)           3
      do 22 k = 1,n                                                      4
         Y(k) = 20.0d0                                                   5
         if(U(k) .lt. 20.0d0 * V(k)) Y(k) = U(k) / V(k)                  6
         W(k) = X(k) / (dexp(Y(k)) - 1.0d0)                              7
22    continue                                                           8
      return                                                             9
      end                                                               10
```

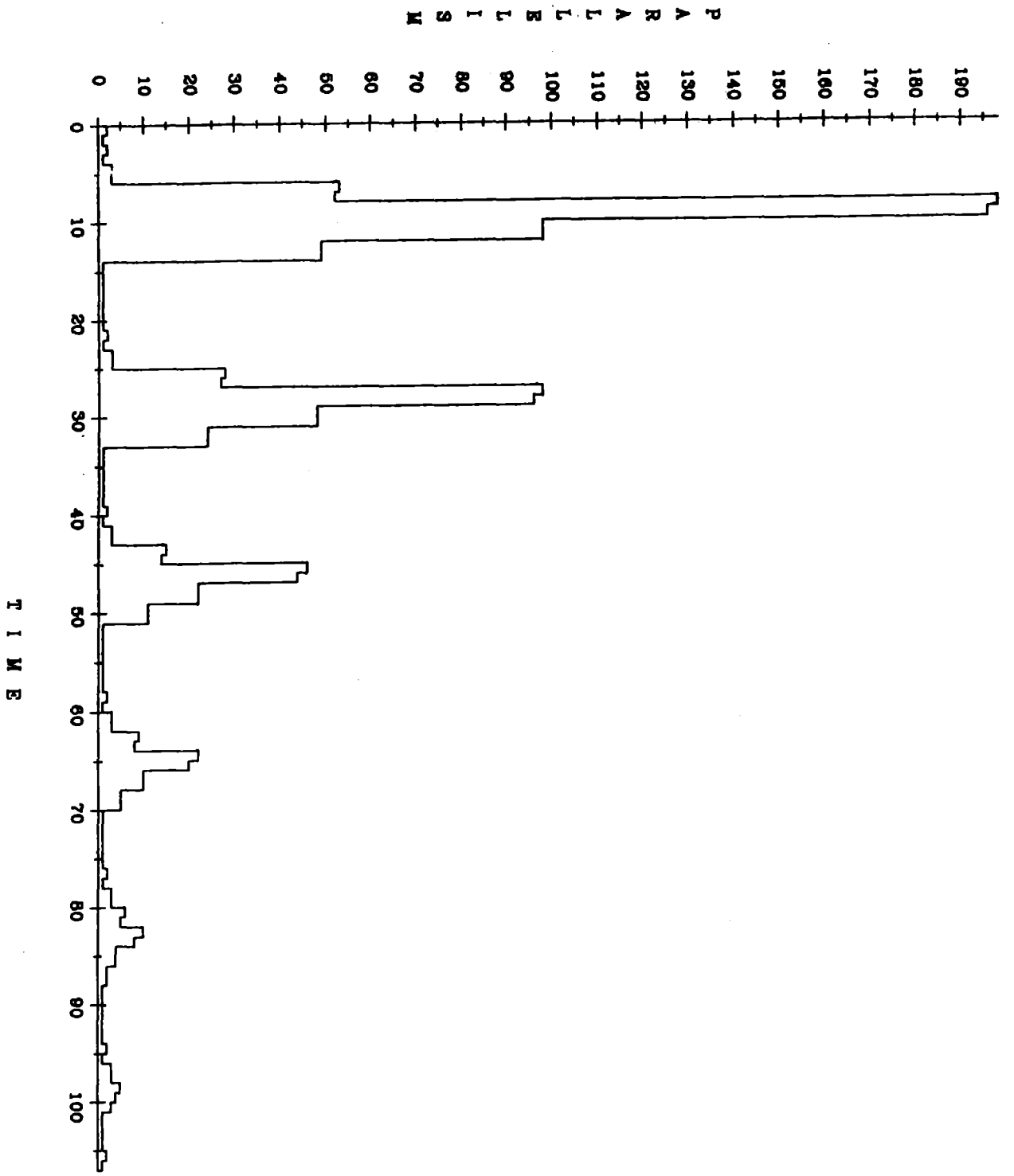
```
      subroutine Loop23(n,ZA,ZB,ZR,ZU,ZV,ZZ)                                  1
      integer j,k,n                                                           2
      double precision QA,ZA(101,7),ZB(101,7),ZR(101,7)                       3
      double precision ZU(101,7),ZV(101,7),ZZ(101,7)                          4
      do 23 j = 2,6                                                           5
      do 23 k = 2,n                                                           6
         QA = ZA(k,j+1) * ZR(k,j) + ZA(k,j-1) * ZB(k,j) +                     7
     .        ZA(k+1,j) * ZU(k,j) + ZA(k-1,j) * ZV(k,j) + ZZ(k,j)             8
23       ZA(k,j) = ZA(k,j) + 0.175d0 * (QA - ZA(k,j))                         9
      return                                                                 10
      end                                                                    11
```

```
      subroutine Loop24(n,max24,X)                                            1
      integer k,max24,n                                                       2
      double precision X(1001)                                                3
      max24 = 1                                                               4
      do 24 k = 2,n                                                           5
24       if(X(k) .lt. X(max24)) max24 = k                                     6
      return                                                                  7
      end                                                                     8
```

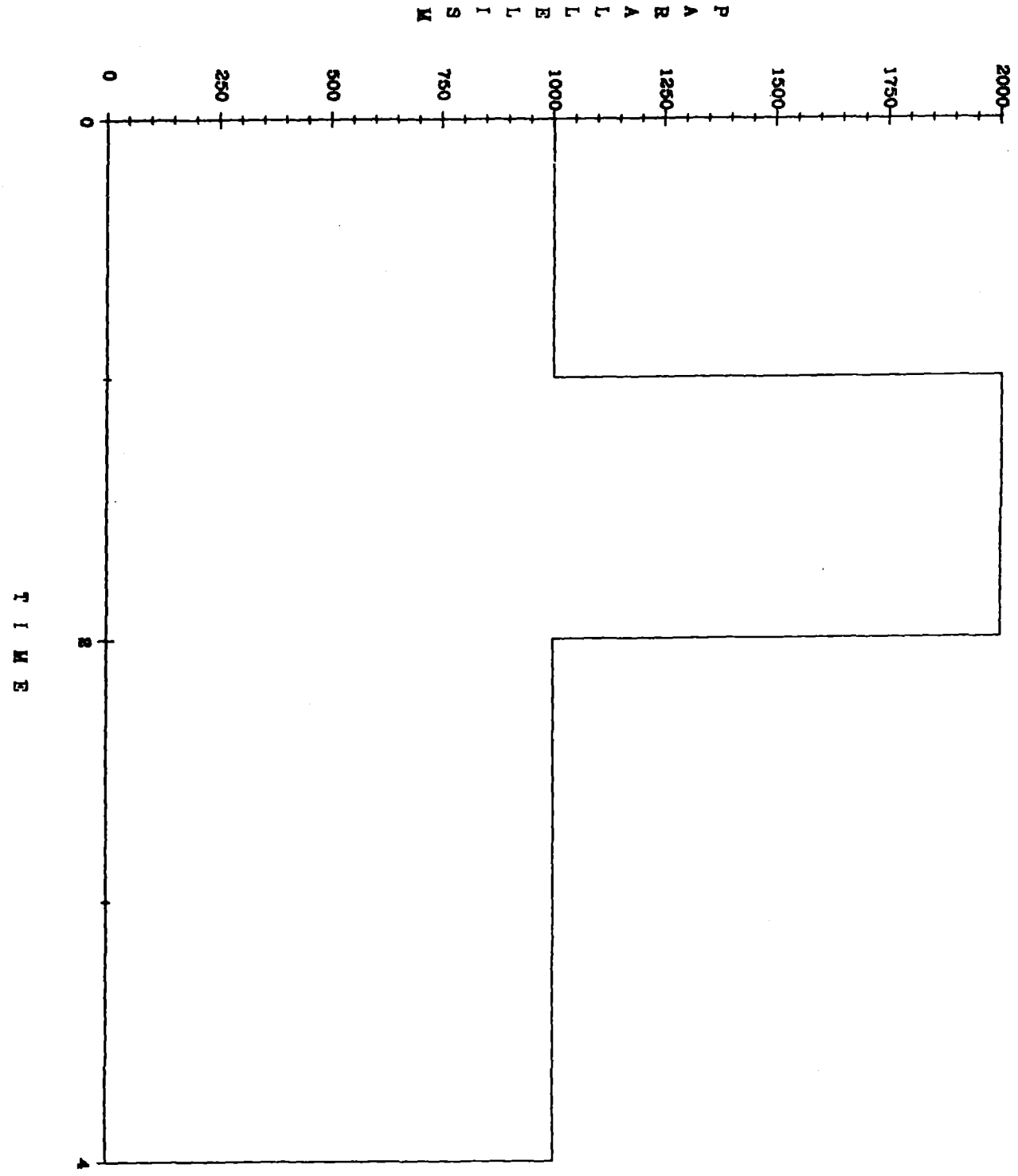
Appendix C: The Parallel Execution Traces



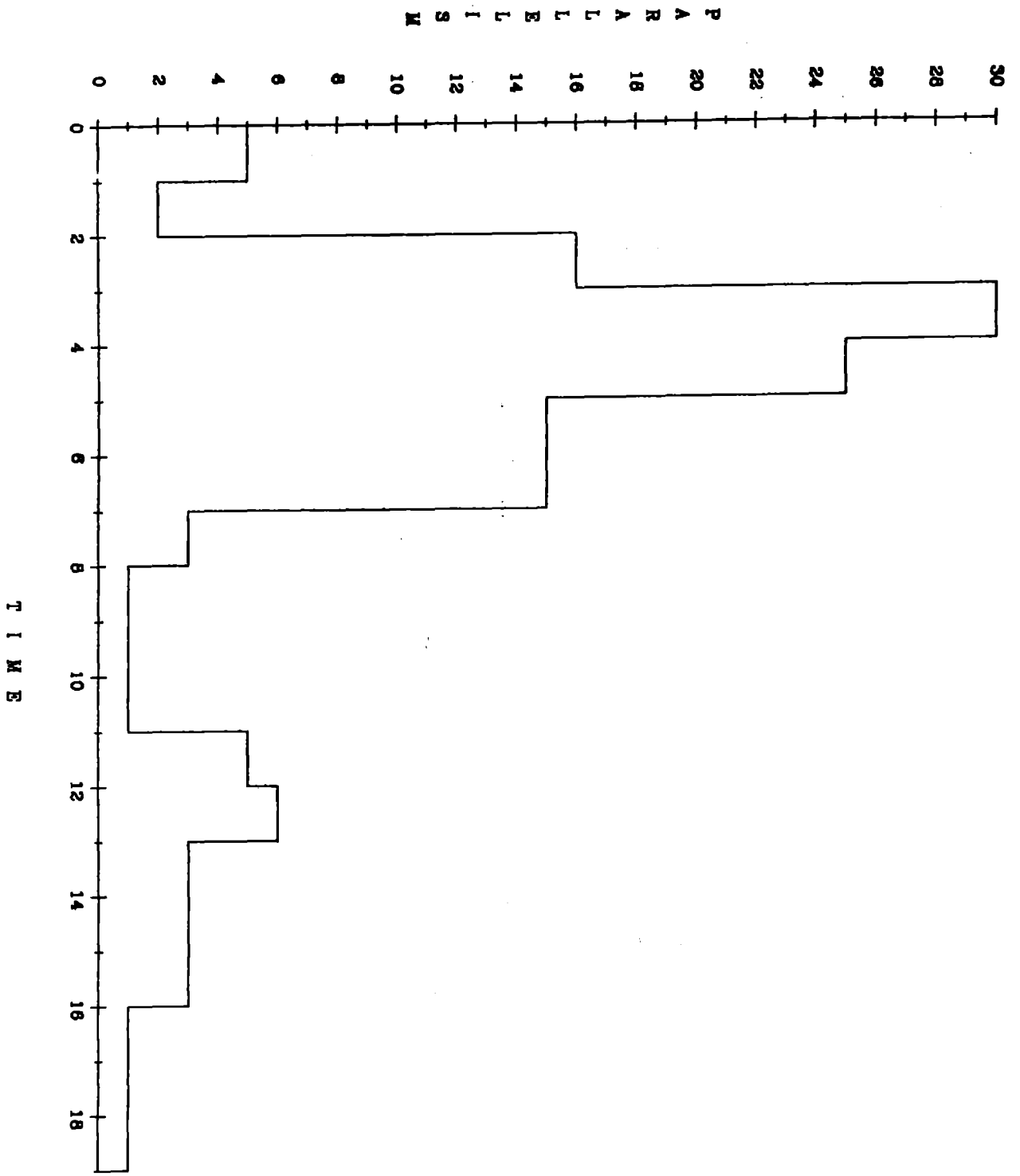
Loop 1 - Hydrodynamic Fragment, n = 990



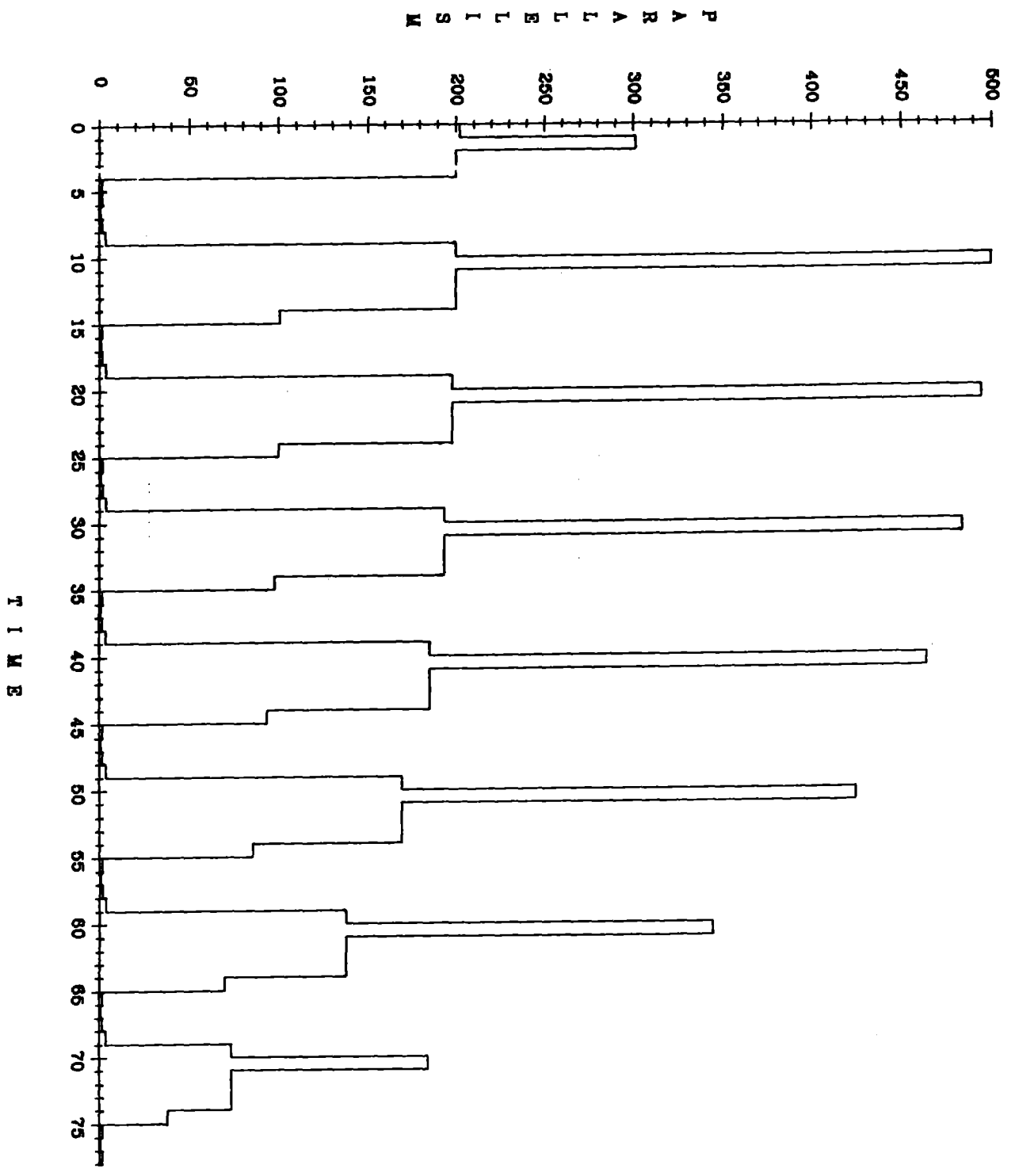
Loop 2 - Incomplete Cholesky - Conjugate Gradient, n = 101



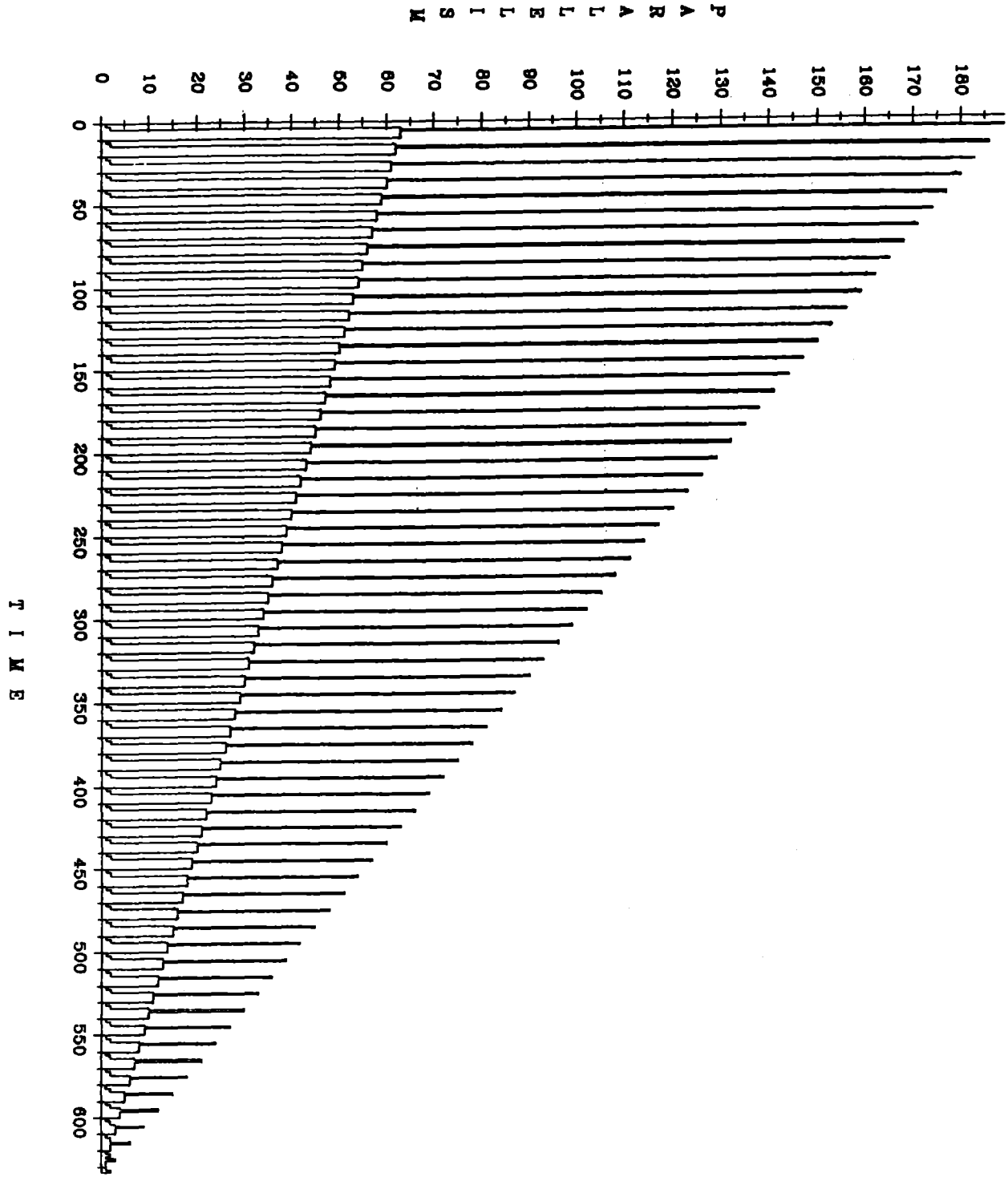
Loop 3 - Inner Product, n = 1001



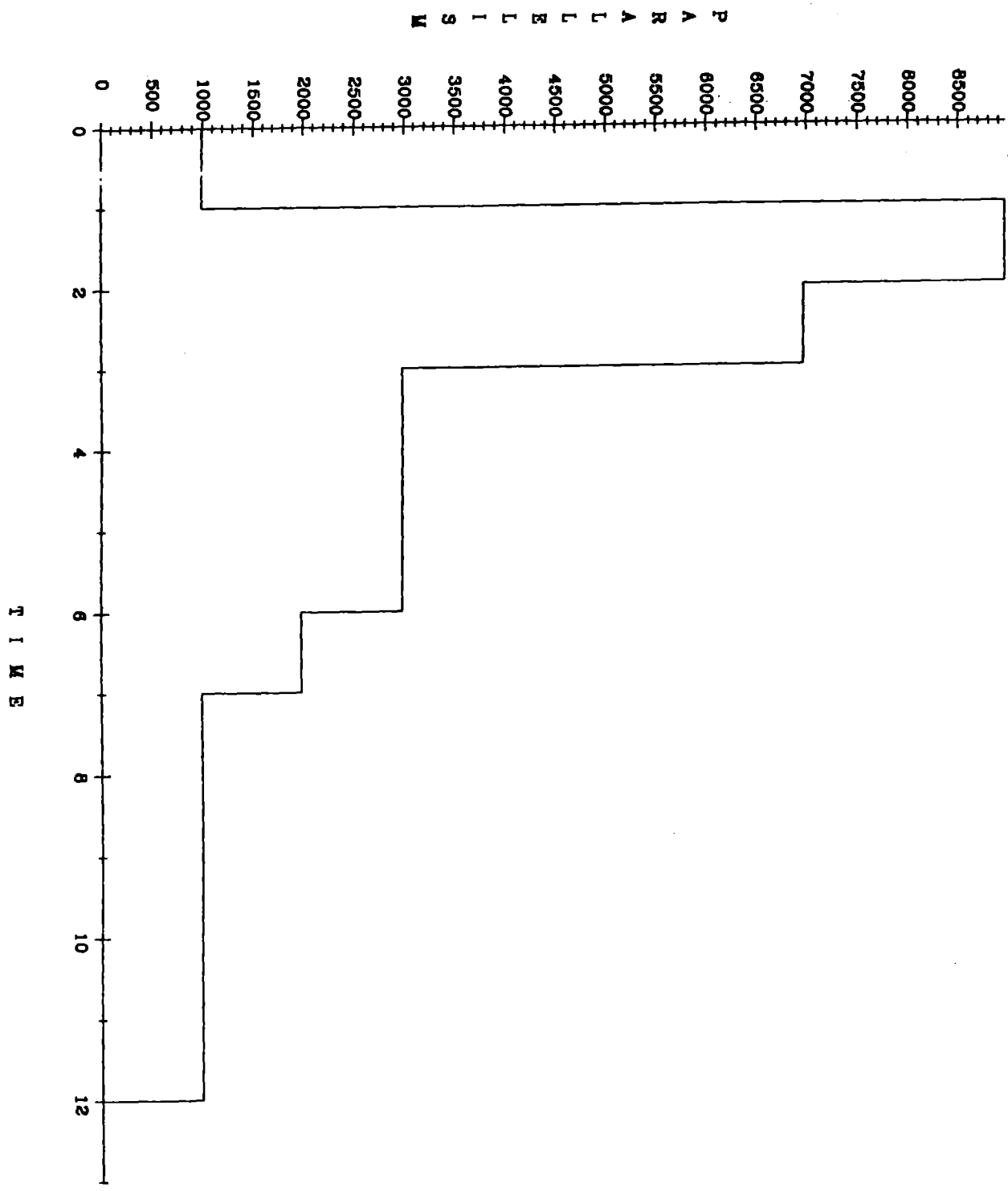
Loop 4 - Banded Linear Equations, n = 35



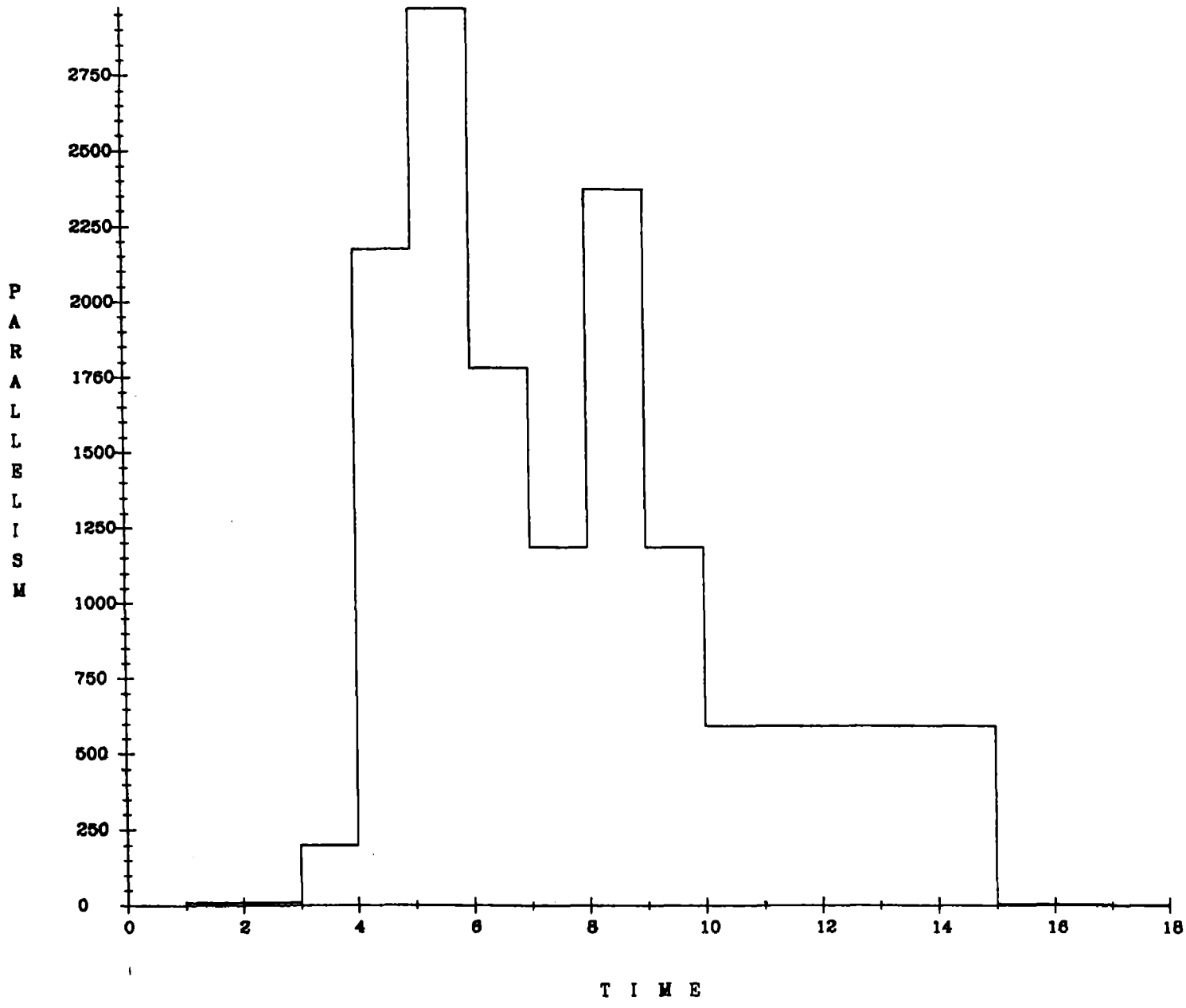
Loop 5 - Tri-Diagonal Elimination, n = 101



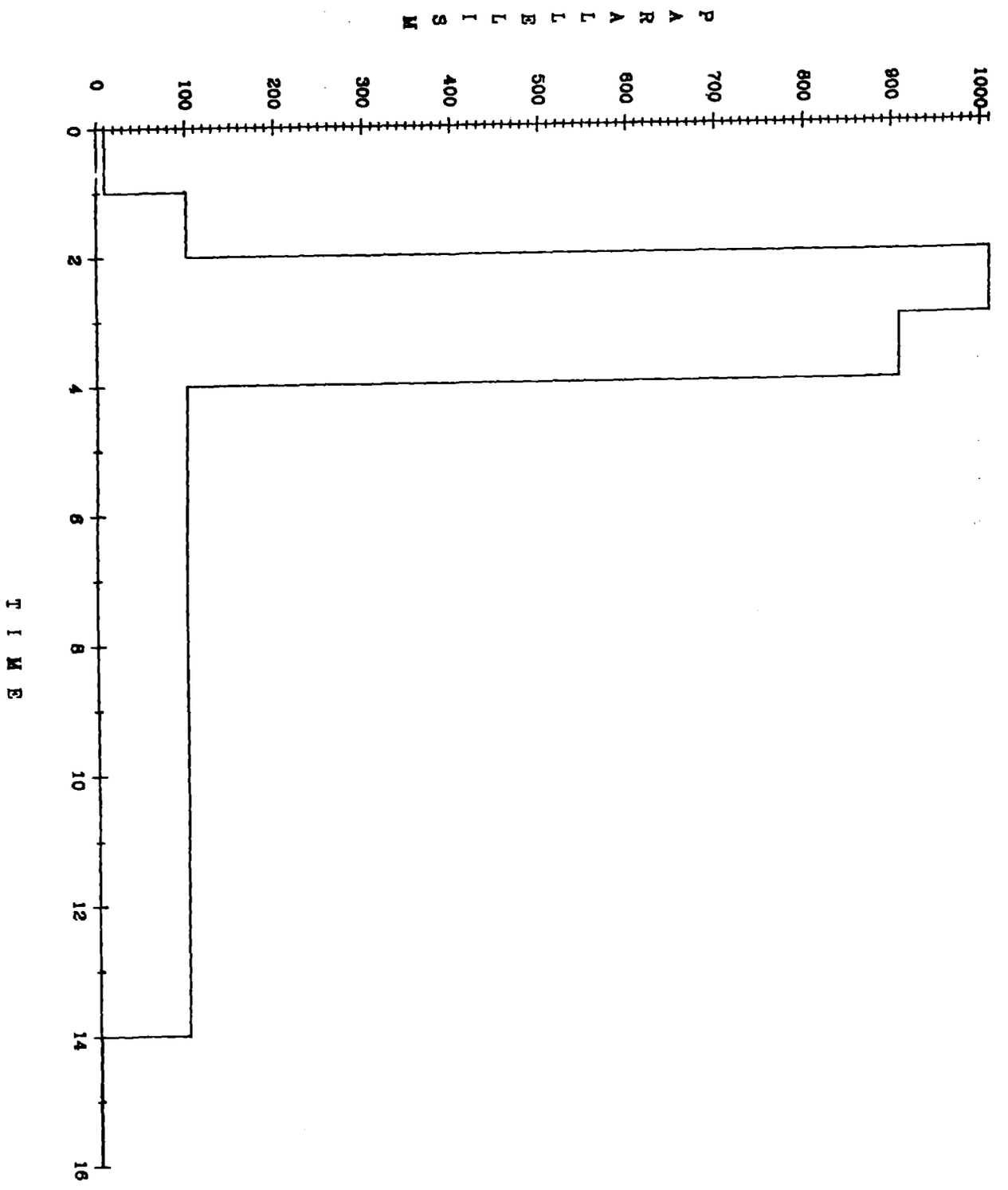
Loop 6 - General Linear Recurrence Equations, n = 64



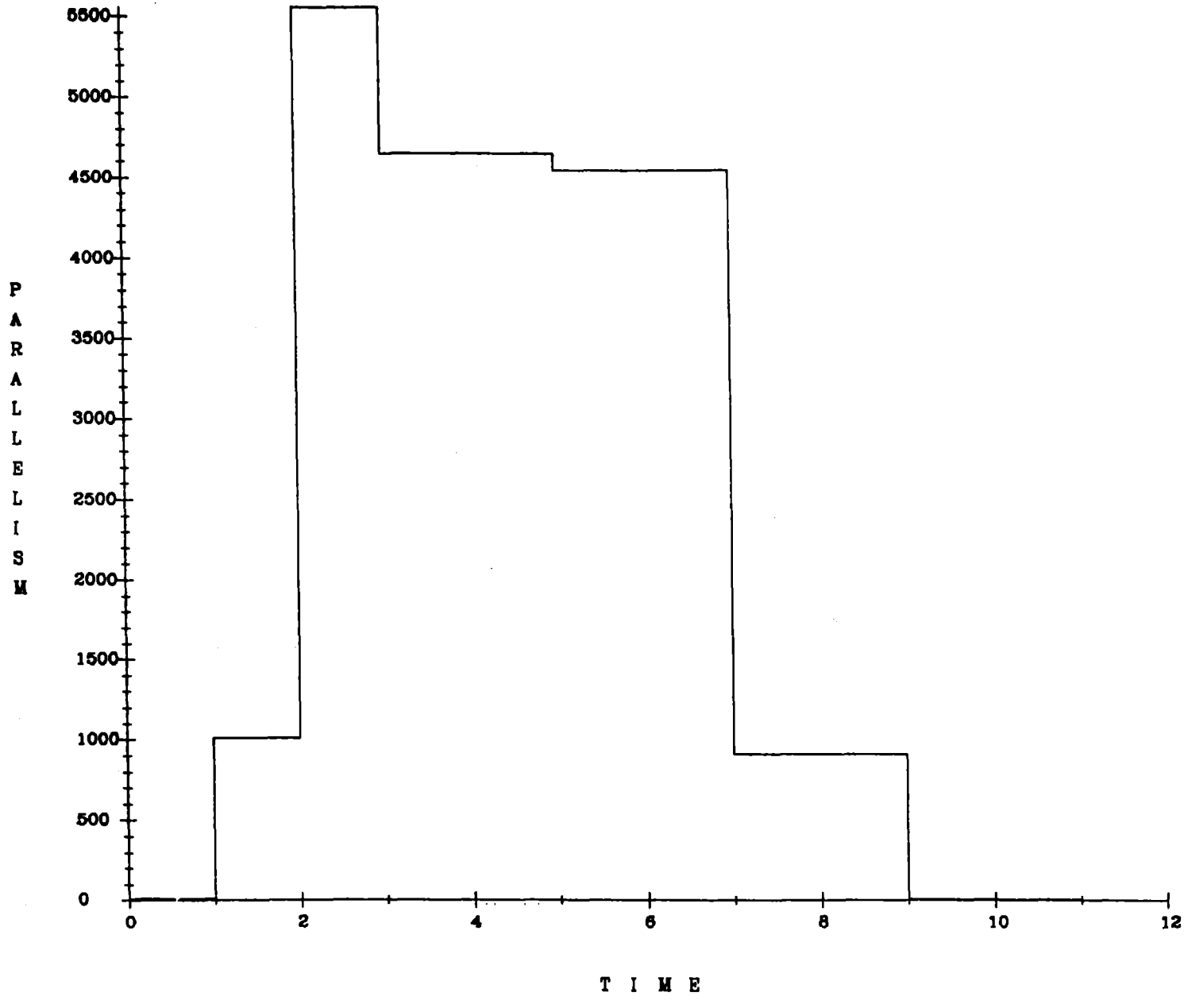
Loop 7 - Equation of State Fragment, n = 995



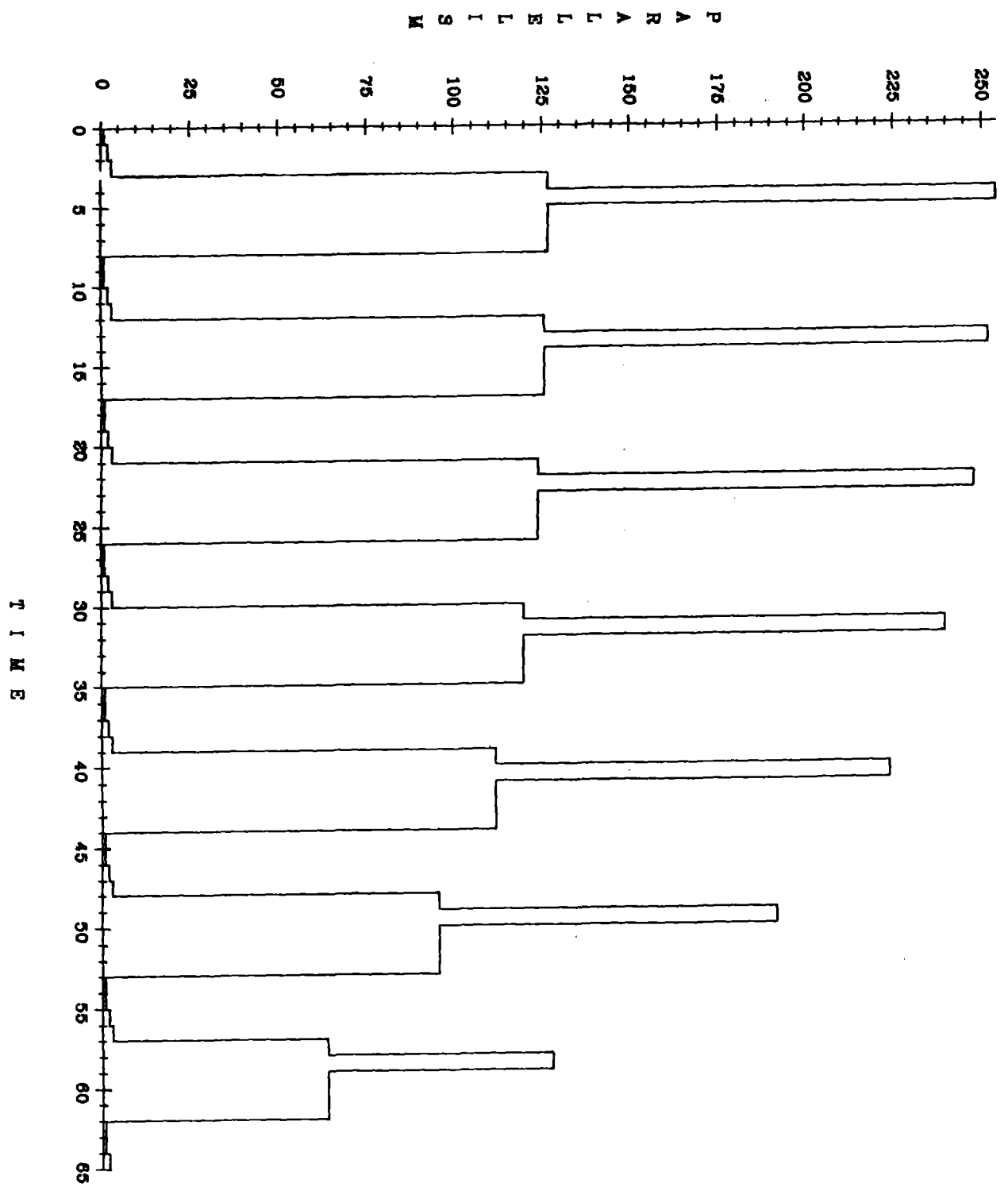
Loop 8 - A. D. I. Integration, n = 100



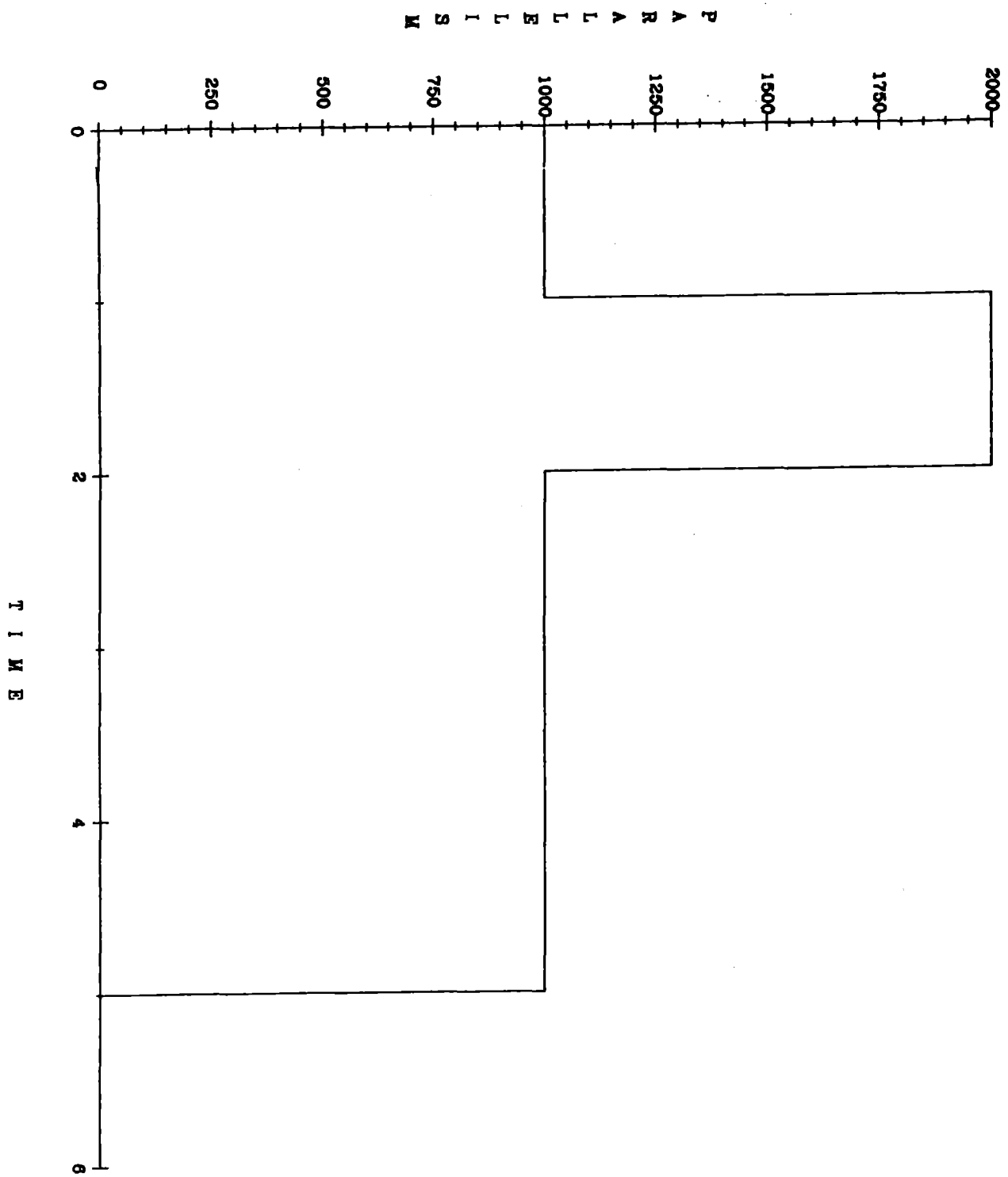
Loop 9 - Integrate Predictors, n = 101



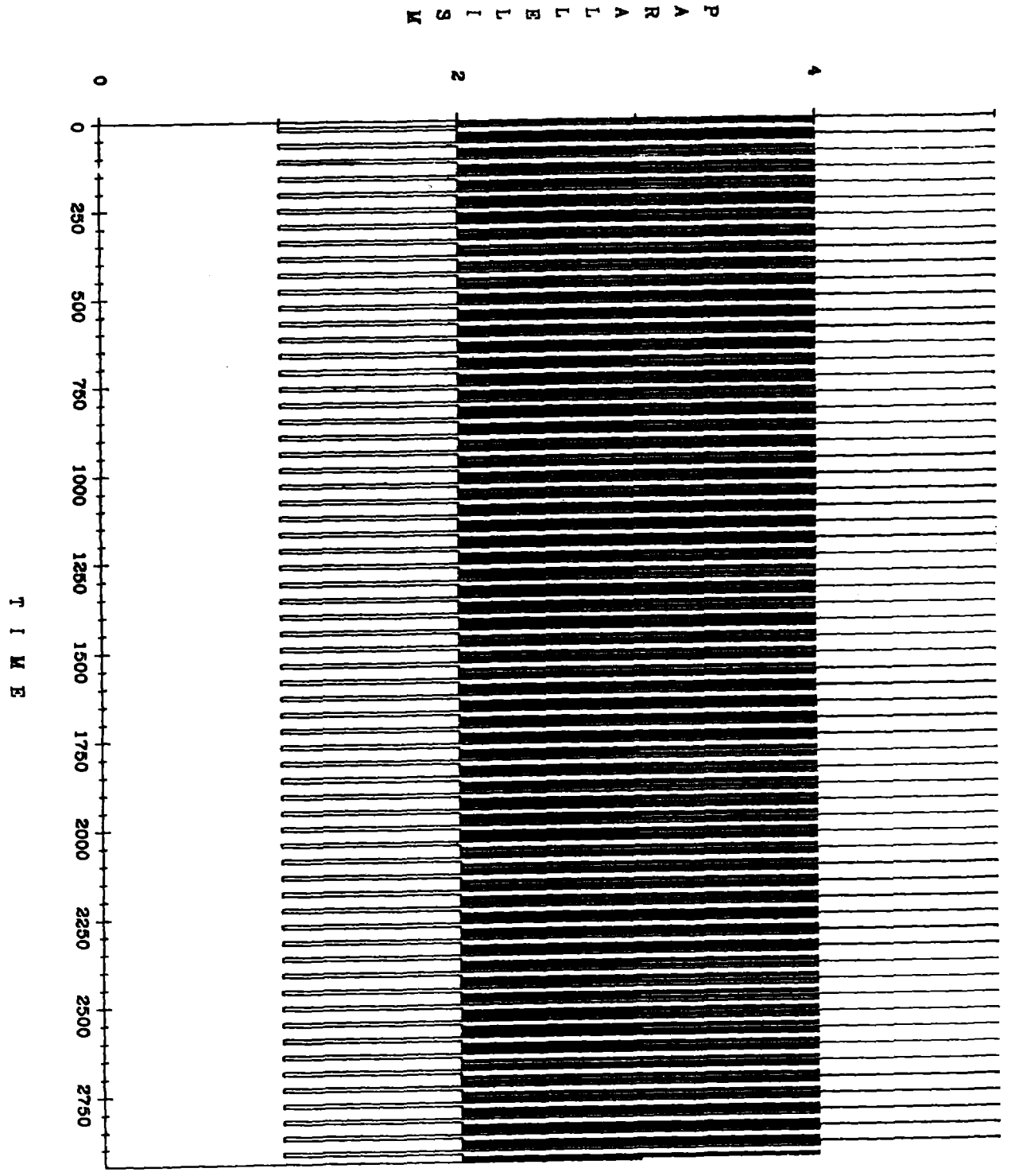
Loop 10 - Difference Predictors, n = 101



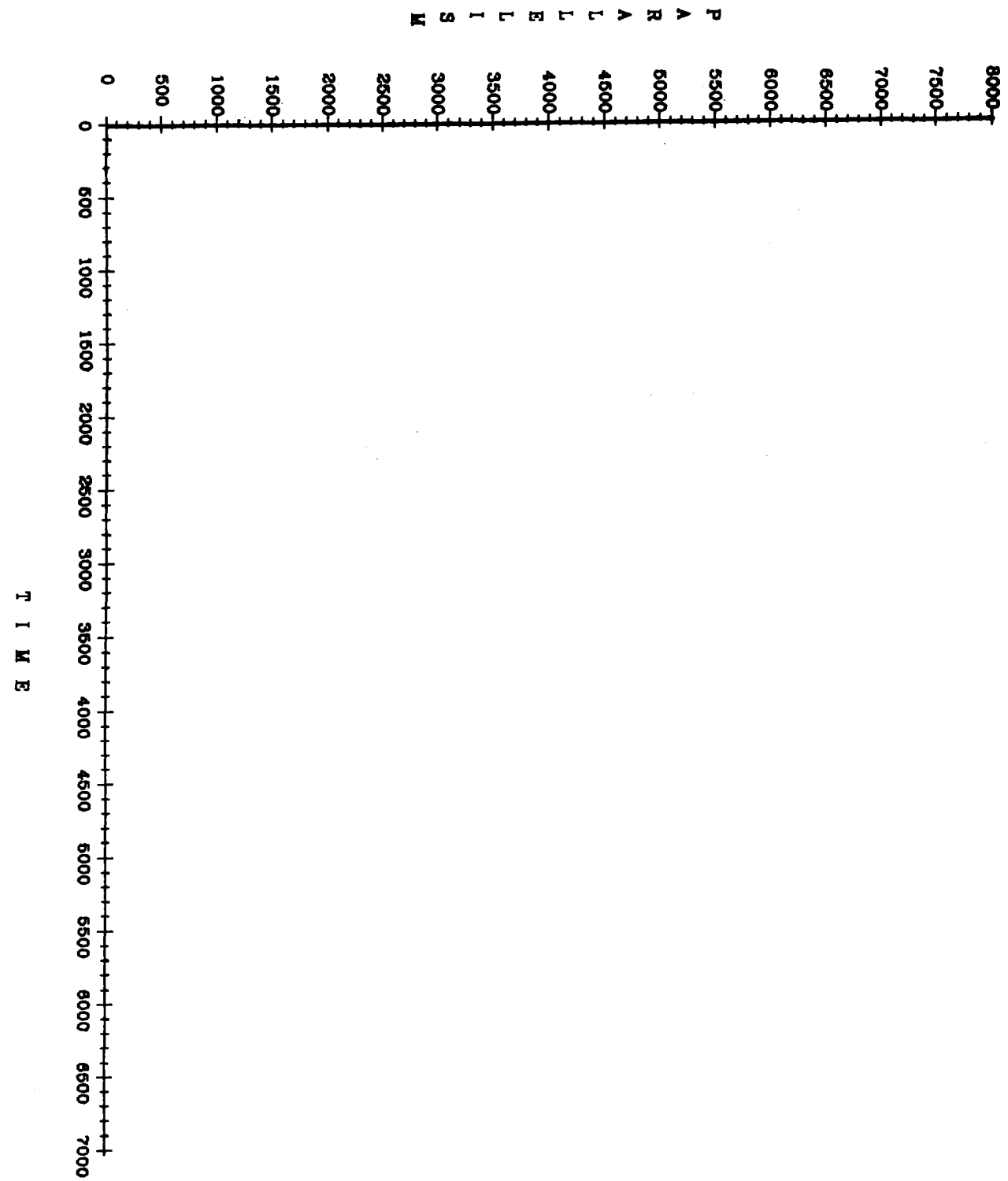
Loop 11 - First Sum, n = 128



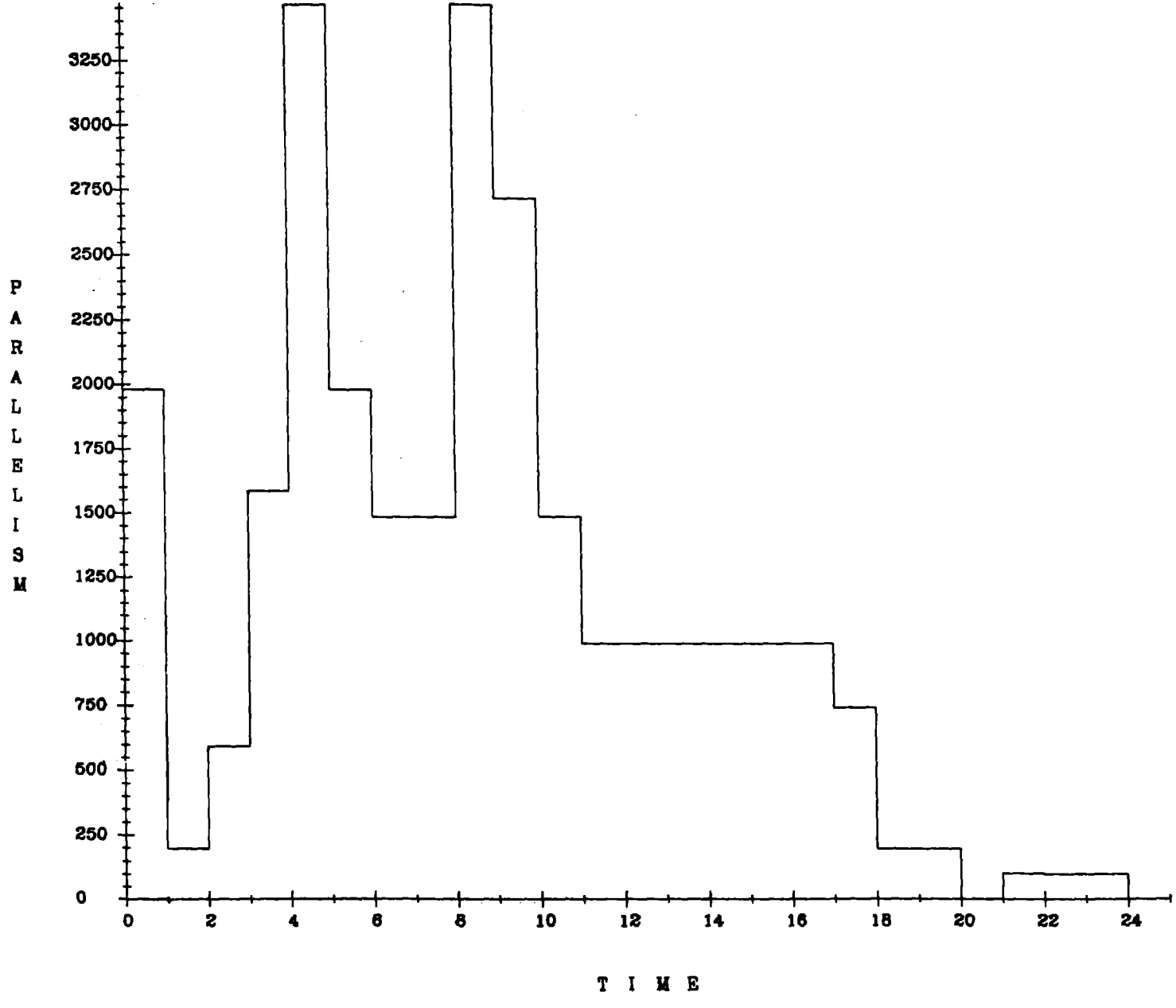
Loop 12 - First Difference, n = 1000



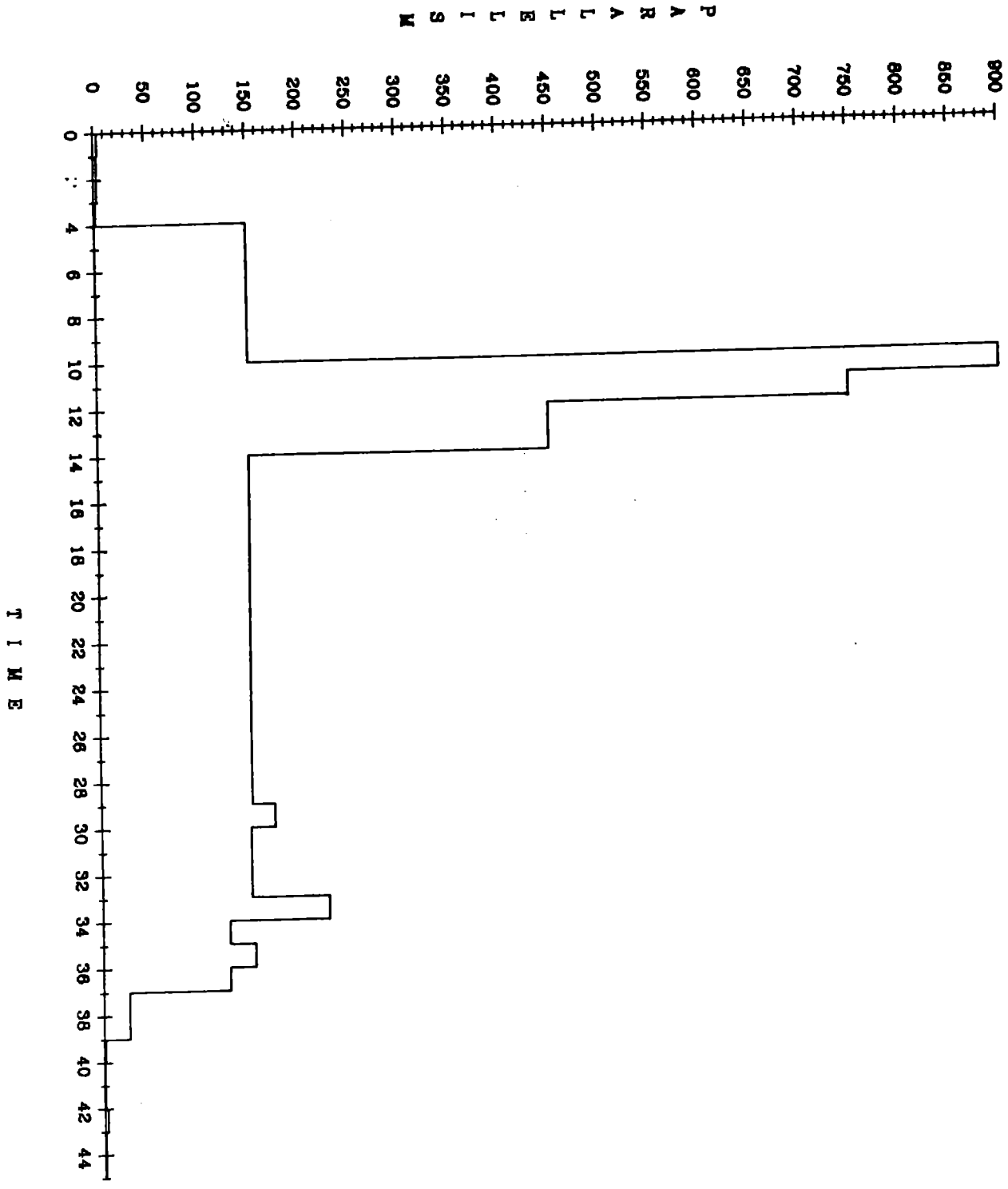
Loop 13 - 2-D Particle in Cell, n = 64



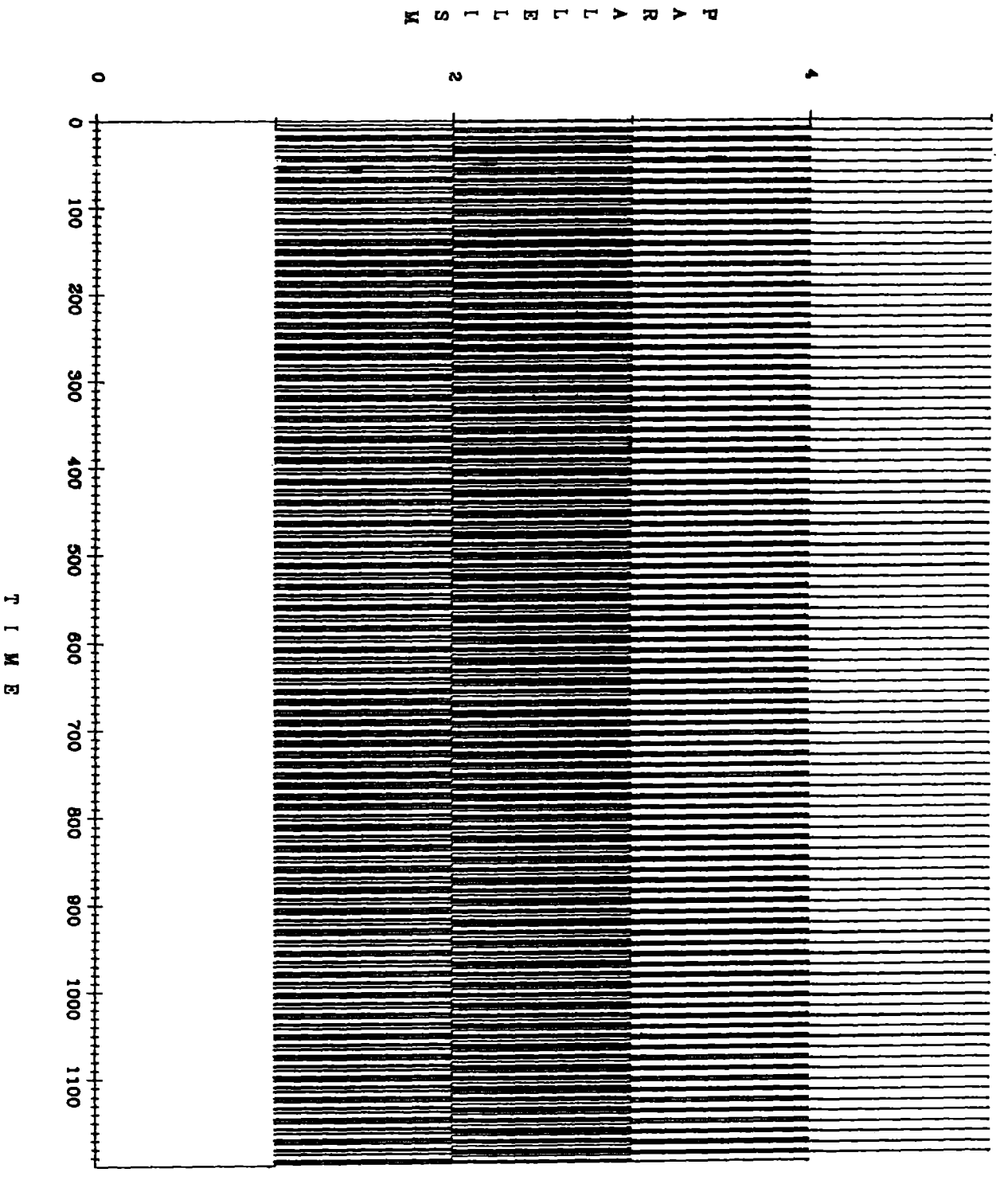
Loop 14 - 1-D Particle in Cell, n = 1001



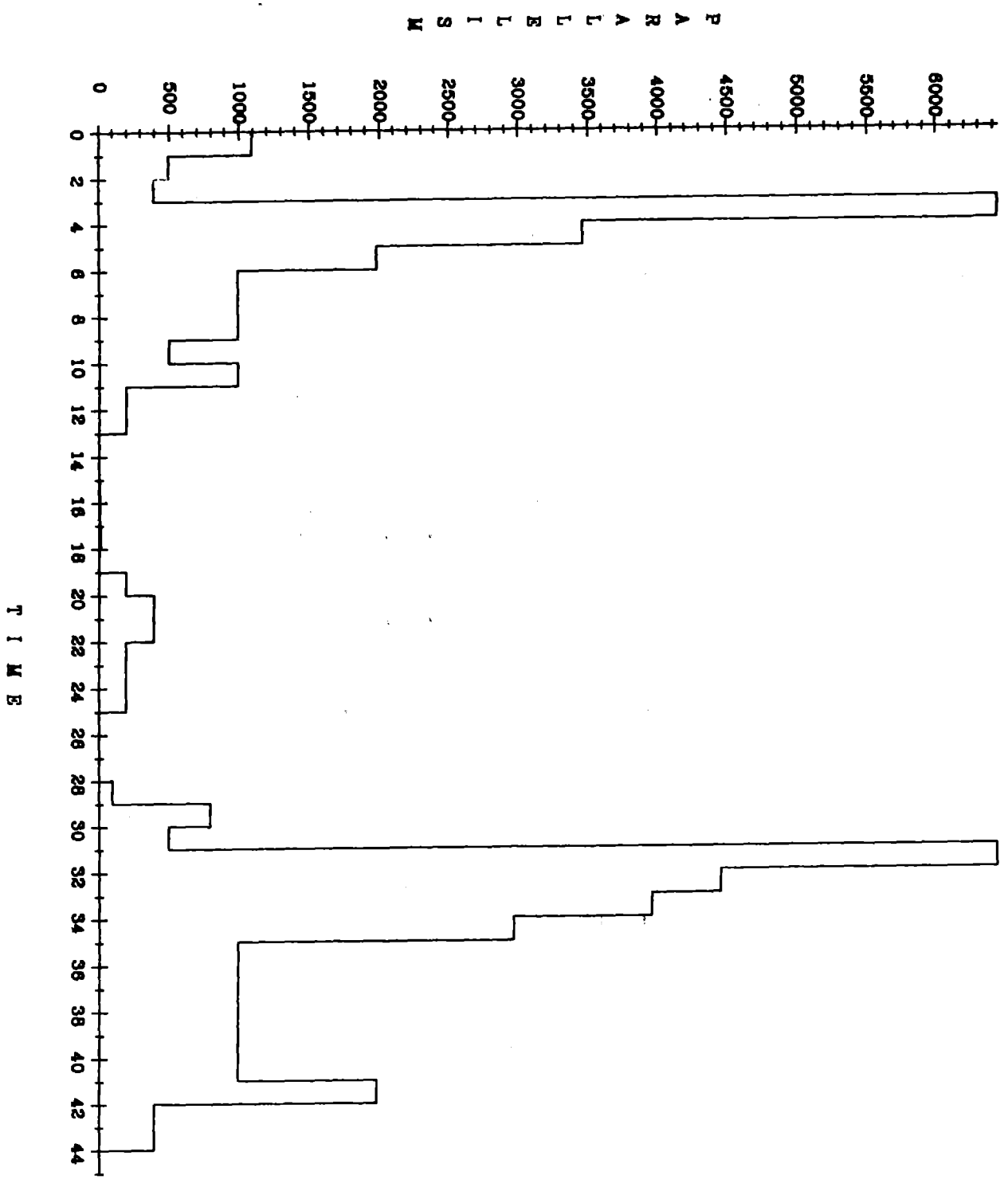
Loop 15 - Casual Fortran, n = 101



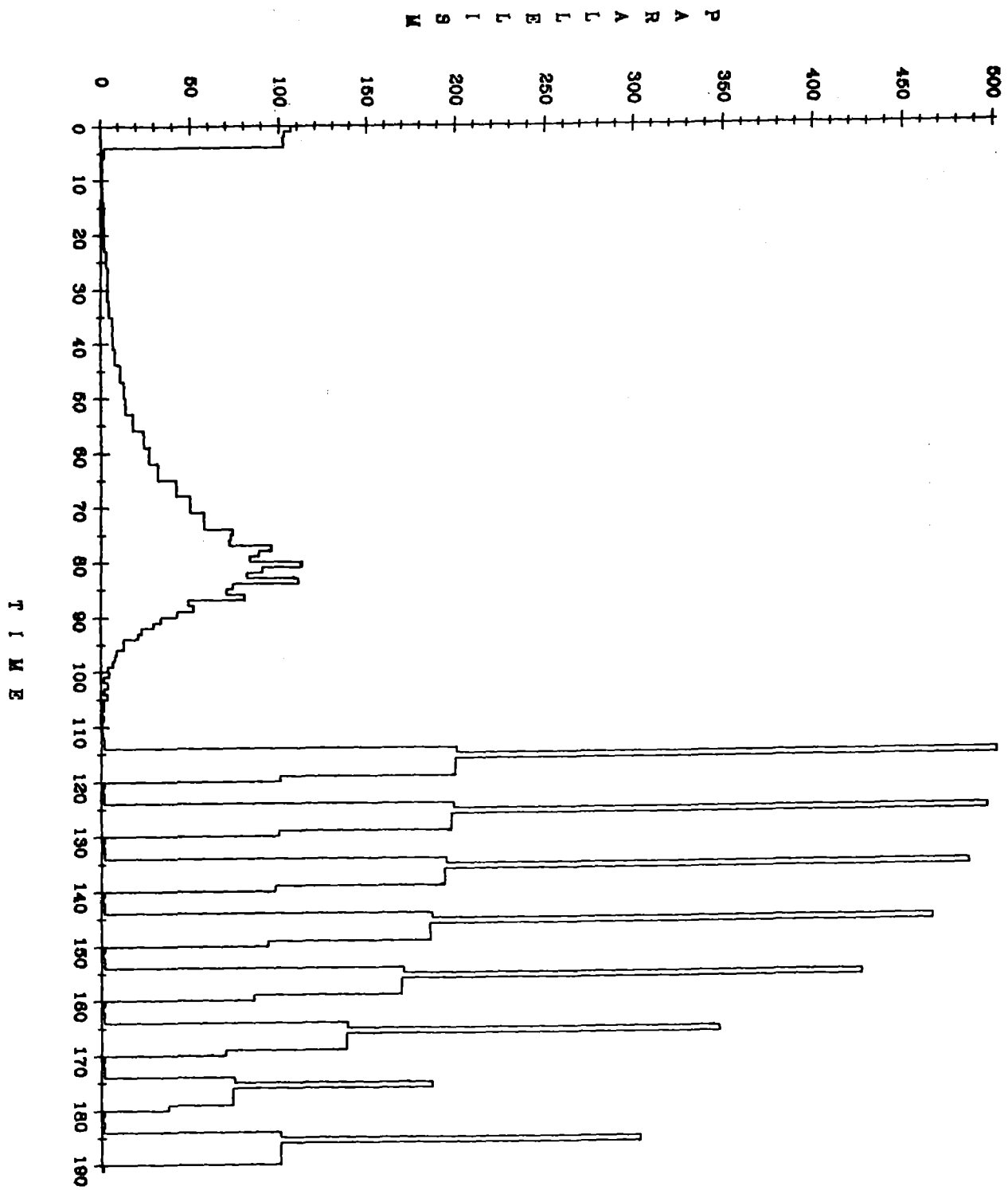
Loop 16 - Monte Carlo Search Loop, n = 75



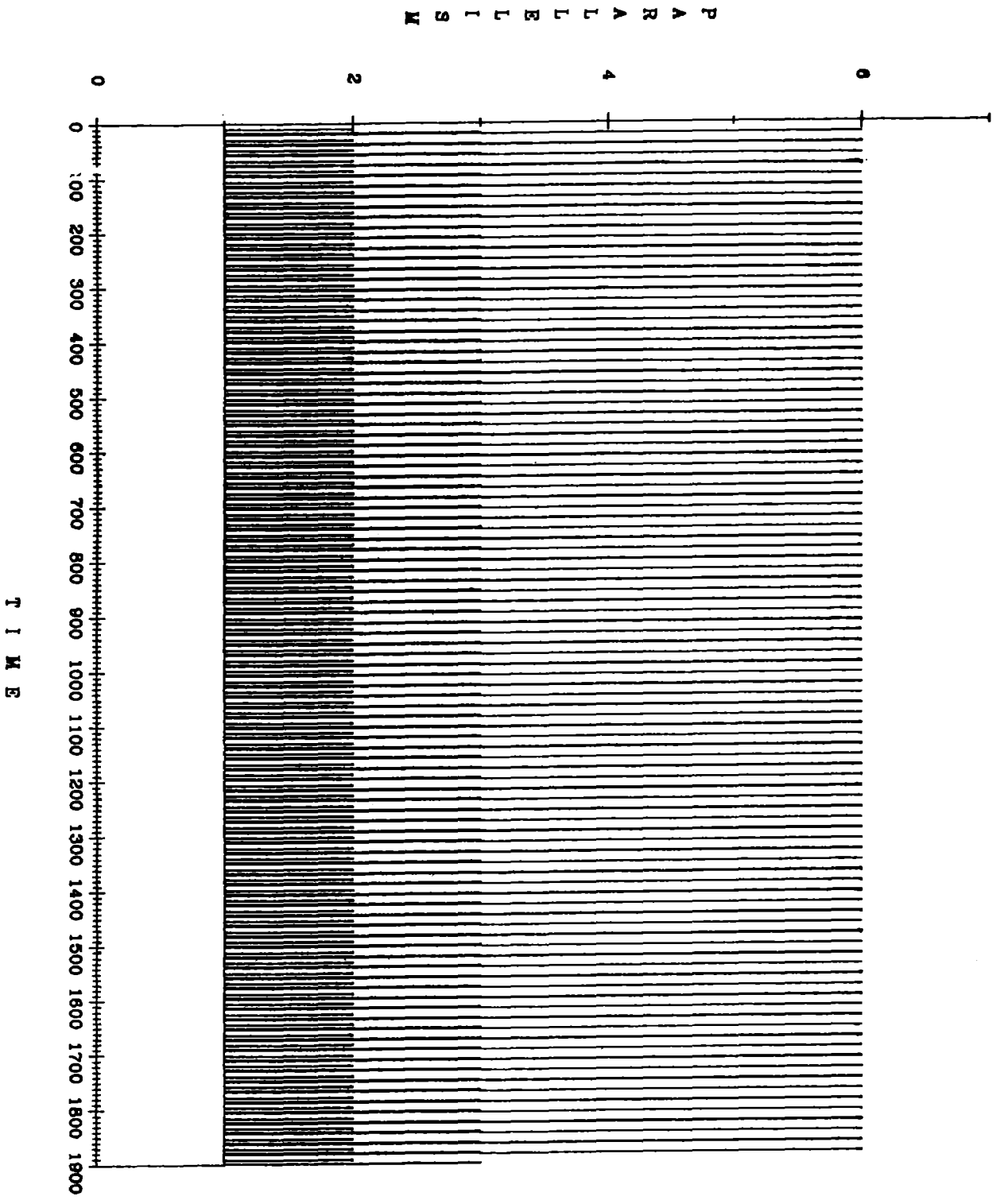
Loop 17 - Implicit, Conditional Computation, n = 101



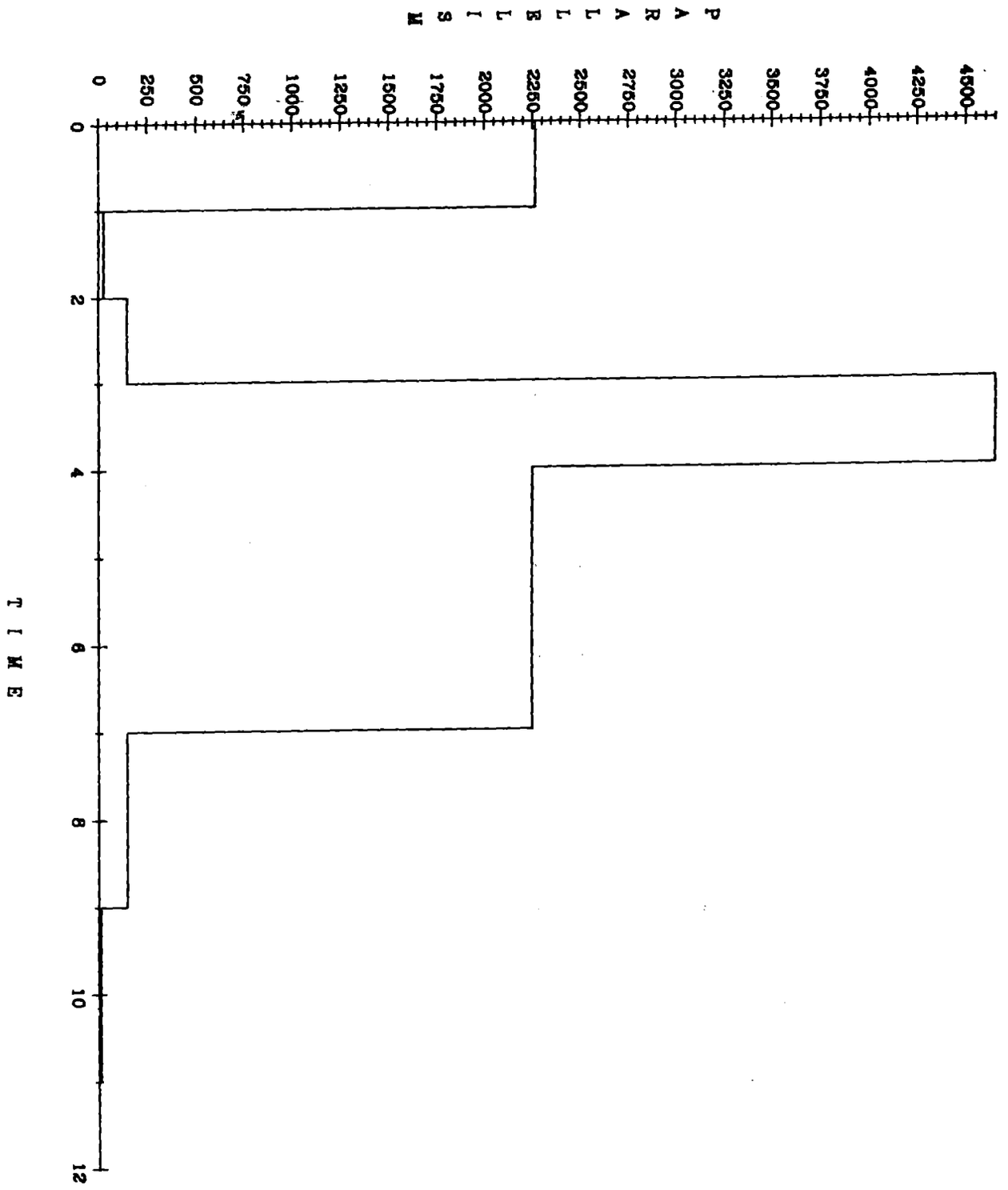
Loop 18 - 2-D Implicit Hydrodynamic Fragment, n = 100



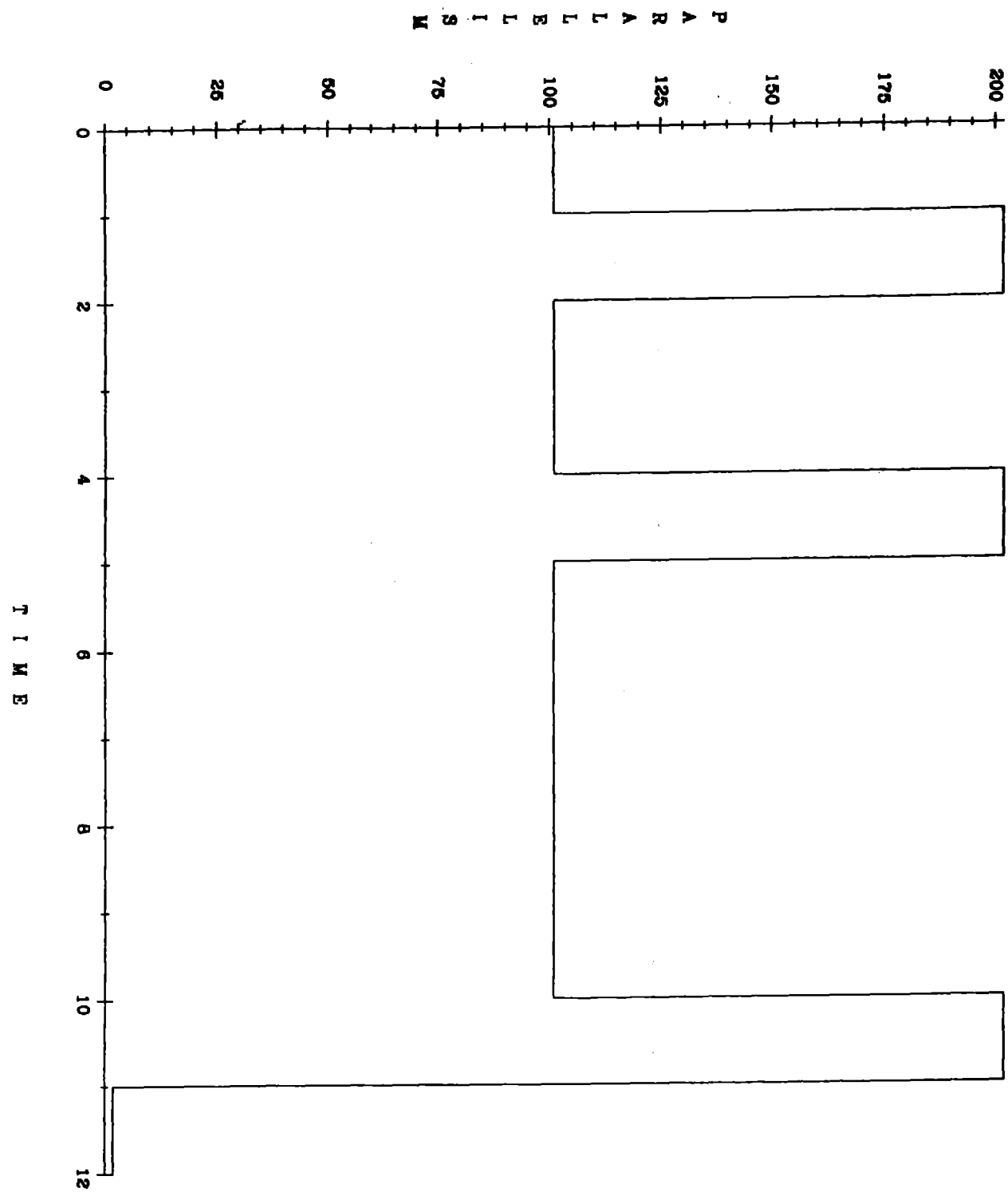
Loop 19 - General Linear Recurrence Equation, n = 101



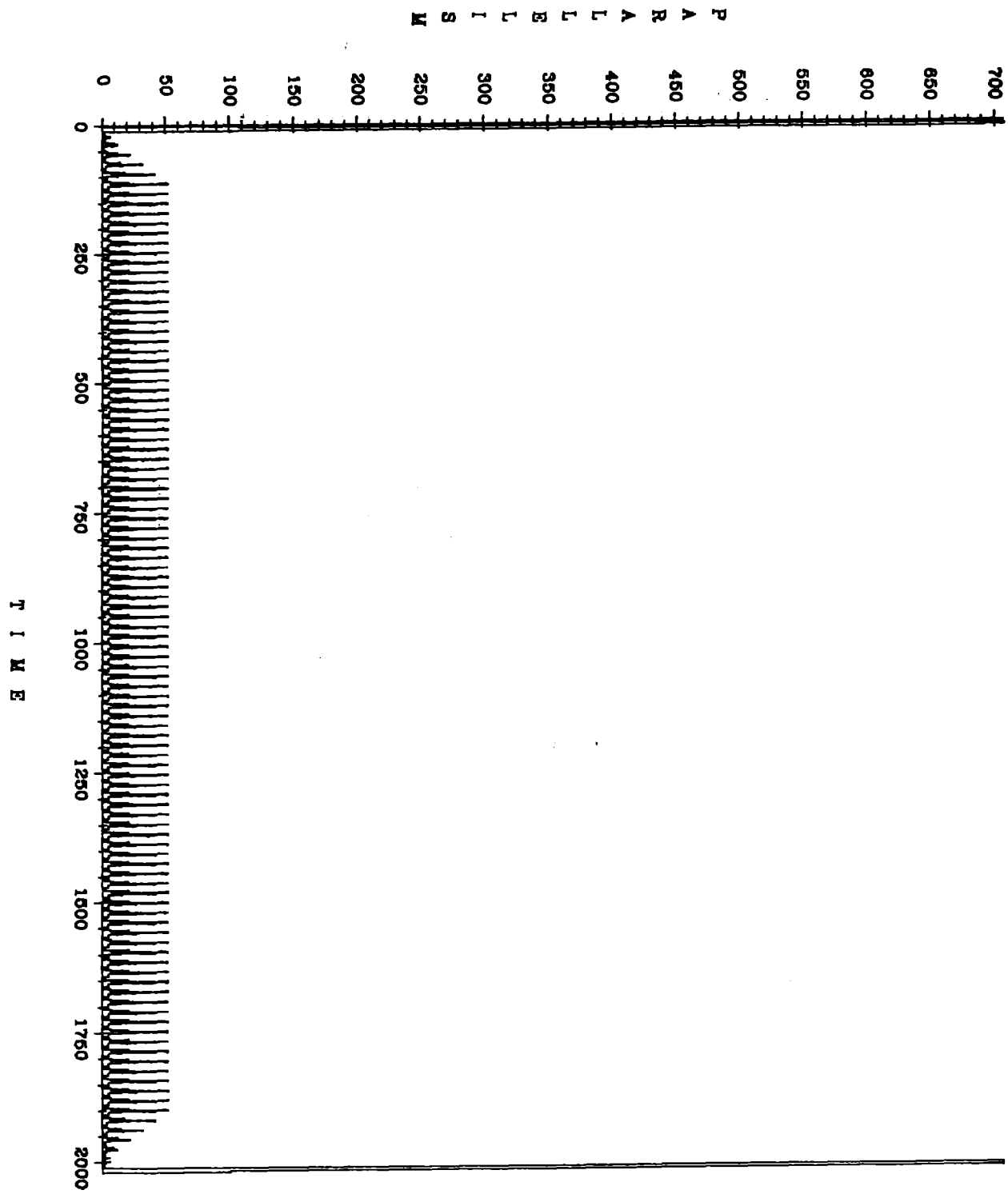
Loop 20 - Discrete Ordinates Transport, n = 100



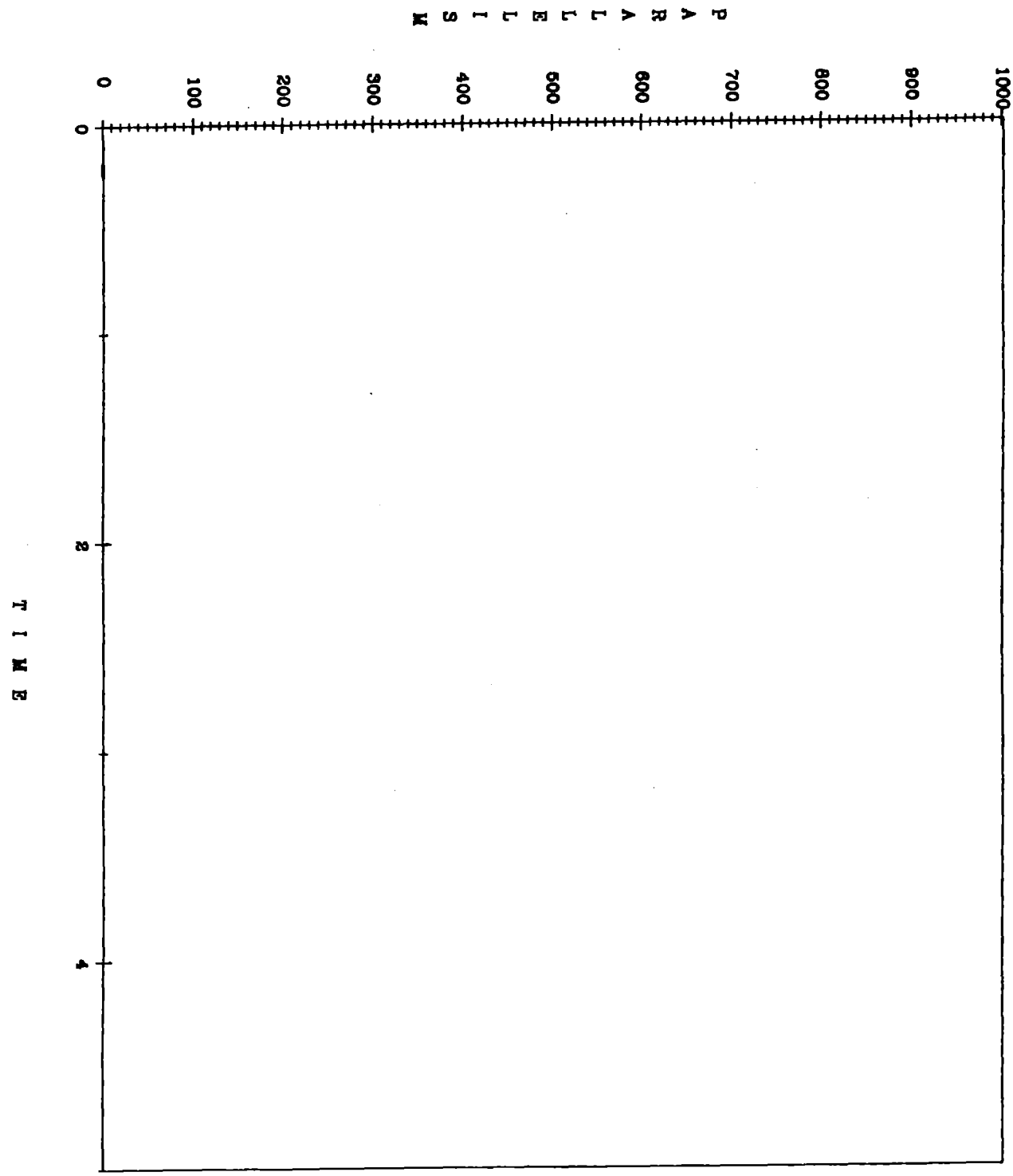
Loop 21 - Matrix Multiplication, n = 10



Loop 22 - Planckian Distribution, n = 101



Loop 23 - 2-D Implicit Hydrodynamics Fragment, n = 100



Loop 24 - Location of First Minimum, n = 1001