MIT/LCS/TR-216

ANALYSIS OF THE SIMPLE CODE FOR DATAFLOW COMPUTATION

John M. Myers

Analysis of the SIMPLE code for dataflow computation

by

John M. Myers, Consultant

May, 1979

This research was supported by Lawrence Livermore Laboratory under contract 8545403.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE

Cambridge Massachusetts 02139

Analysis of the SIMPLE code for dataflow computation

John M. Myers

ABSTRACT

We analyze a problem in hydrodynamics from the standpoint of computation on a dataflow computer that is not yet fully specified, with the objectives of helping to further specify the computer and helping to develop VAL as its source language. Lawrence Livermore Laboratory supplied the algorithm for hydrodynamics, including heat flow, as a 1749-line FORTRAN code called SIMPLE.

The algorithm viewed as 'abstract' (i.e. independent of physical arrangements in space and time for its realization) is shown to imply spatial and temporal structure that must appear in any and all implementations. Both for hardware design and program compilation it is useful to map this structure to grosser levels of description, with the grosser levels reflecting modularity of computational resources conjoined with modularity of the algorithm. Following Holt (1979) we use role diagrams to display spatio-temporal structure at different descriptive levels, so as to guide translation into VAL as well as the analysis of the time to compute.

Inter-resource communication essential to the problem is displayed, and various issues of machine design are defined. Using VAL with one set of extensions, we express the algorithm so that in principle it can be compiled for execution by a dataflow computer. Input-output functions beyond those implied by the SIMPLE code are discussed. A second set of extensions to VAL is advocated to express the conjunction of problem and resource modularity, so as to guide compilation. The dependence of time to compute on the number of processing units is shown for various aspects of the problem.

KEYWORDS: DATAFLOW, ALGORITHM ANALYSIS, PARALLEL COMPUTATION,

COMPUTATIONAL HYDRODYNAMICS, ROLE DIAGRAM.

Analysis of the SIMPLE code for dataflow computation

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

Analysis of the SIMPLE Code for Dataflow Computation

1. Introduction: Hydrodynamics Meets a Dataflow Computer

The equations of physics are prescriptions for calculating; from some presumed starting conditions, they generate a "future". The calculation of this "future" involves many events, each of which "consumes" items -- values of variables -- and "produces" other items. Because an item cannot be consumed before it is produced, these events are subject to constraints of sequencing. These constraints impose a pattern on the calculation.

Although the equations of physics constrain the calculation, they do not fully determine it. The pattern is partly determined also by the method of solution employed and by the structure of the computer. Thus the same (partial differential) equations can result in different patterns of calculation, according to the method of solution and the arrangement of computational resources. For this reason the pattern of computation for a given type of problem, say hydrodynamics, evolves as methods and computational resources evolve. Pattern, method, and resources are coupled in their evolution, with each selected in part to support and to draw on the others.

Over most of history the computer (human or machine) had only a sequential processing capacity, so that computation was necessarily performed one step after another. Thus methods which emphasize concurrency were not called for, and as a result are today relatively unexplored and undeveloped. Not only computers, but also numerical methods have evolved in a context that is weighted toward the sequential, and away from the concurrent.

Via such means as dataflow architecture (see Dennis, 1978), an increase in speed can be brought about by an organization of computational resources that allows concurrency of many events. This report is concerned with fitting -- or refitting -- a pattern that evolved in a sequential context onto a dataflow computer. The report is based on a case study of an example program written in FORTRAN for a sequential machine for the solution of a problem of hydrodynamics, including heat flow. This program was prepared by Lawrence Livermore Laboratory, and is named SIMPLE. The initially presented questions were:

1.1) What is involved in translating the SIMPLE program from FORTRAN (suitable for a sequential computer) into a dataflow language (the VAL language in particular); and

1.2) Compared to a sequential computer, what speed advantage can be expected from a dataflow computer in the execution of the SIMPLE program?

To realize the potential advantage of a dataflow computer, its program must be free of unnecessary sequencing constraints. Sequencing constraints come from many sources, and their necessity depends on ones point of view. Primarily we report on the narrow view that sees sequencing constraints as imposed by the data dependencies of the FORTRAN program. In this view the "translation" per item 1.1 entails the removal of sequencing only as far as possible without disrupting the data dependencies expressed in the FORTRAN program. Such a translated program would be expected to produce numerical results identical to the FORTRAN program, apart from round-off errors.

But the narrow view fails to:

a) realize the potential for advances in speed, and

b) open the physics itself to new perspectives made possible by the power to express concurrency.

Although their resolution is outside the scope of this report, we shall define some broader issues of solution methods, machine design, and physics.

With respect to item a), the translated program will still contain unnecessary sequencing constraints, imposed by a method of solution of the equations of physics. For example, the back-substitution method (Crowley, Hendrickson and Rudy, 1978) for solving the implicit formulation of heat flow does not realize the potential of dataflow architecture, and it appears that a method could be developed that (for a dataflow computer, but not for a sequential computer) would be substantially faster. Thus in presenting our results, we shall distinguish sequencing constraints that come from the happenstance of the numerical method embodied in SIMPLE from constraints that come from less malleable sources.

Once the method of solution is considered as variable and not fixed, issues of machine design surface. If methods and machine are to be developed in concert, it might be best to tailor the machine to a certain class of methods, to the detriment of its performance with methods outside that class. If the dataflow computer is seen as a network of interconnected processors, then this issue arises with respect to the communications facility that provides processor-to-processor communication. The problems under study stem from spacially distributed fields that interact in a <u>purely local manner</u>. From this locality one can show that the equations can be solved on a dataflow machine using a communications network which directly links only nearest neighbors, so that a "global" communications facility is not required. Local networks are cheaper and faster than global networks; however the methods that they support have drawbacks with respect to speed, so that the question of <u>local</u> vs. <u>global</u> remains open. One way of posing the issue is through the following question:

1.3) What number N' of globally connected (i.e. fully connected) processors have the same cost as N locally connected processors, under the condition that the total memory of the two configurations be the same?

The idea is that the speed loss from the restriction to local connectivity might be regained through the use of a larger network of processors. In other words for a given investment there is a trade-off between fewer fully connected processors and more locally connected processors. If these two contrasting configurations are to be evaluated in their performance on a given problem, then total system memory should be the same for each configuration.

With respect to item b) it may be of theoretical interest to introduce a class of dataflow computers to model <u>what is meant</u> by the equations of physics.

2. The Hydrodynamic Fields

Given finite propagation velocities, the fields defined by the equations of physics can be pictured, as they were by Huygens, as networks of communicating entities, all operating concurrently. A partial differential equation represents a limit as the network becomes progressively more fine-grained. Computation is possible, however, only if the limit is not taken, or if it is "undone".

Via one or another numerical method the partial differential equations are transformed to difference equations defined on a spatial mesh of N zones, with each zone have corners at nodes, as shown in Fig. 1. In terms of the parameters defined in SIMPLE, one finds

$$N = (LMX-LMN)*(KMX-KMN) . \qquad \text{(Eq. 2.1)}$$

SIMPLE employs a Lagrangian formulation, in which the mesh is deformable; each node is thought of as a "tagged atom", carried along in a fluid whose motion is described by the difference equations. By extending the discussion of Morse and Feshbach (1953, vol 1, p.847-8) to equations of hydrodynamics, one sees Huygen's principle works on a sufficiently small region of the mesh. For a given node, one can choose an enclosing curve through the zones that bound it, and with the result that, by interpolation, the acceleration of the node depends only on the properties of the zones that bound it. A similar argument could lead to the conclusion that the current properties of a zone depend only on past poperties of the nodes at its corners, but SIMPLE is based on a variation of this argument. Properties such as pressure and density are defined only for zones and not for nodes, and the current properties of a zone are shown to depend on their past values together with the current deformation of the zone, along with the current rate of deformation of the zone.

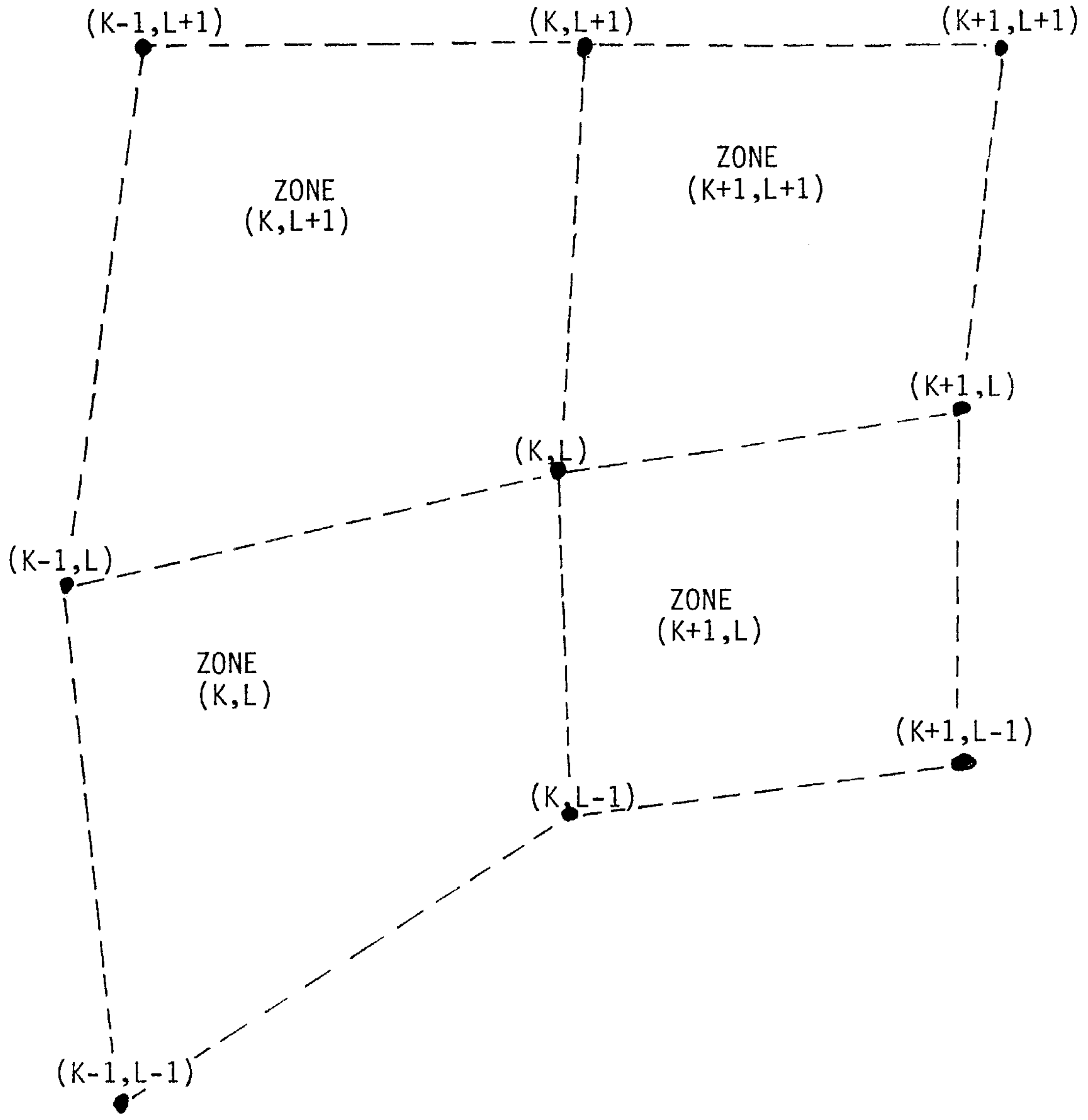


Figure 1: Nodes (shown as heavy dots) and zones (enclosed by dotted lines).

The main fields are defined by Crowley, Hendrickson, and Rudy (1978) as follows:

Field	Zonal name in FORTRAN	Definition
ε	E	energy per unit mass
p	P	pressure
q	Q	artificial viscosity
ρ	RHO	density
θ	TEMP	temperature
τ		specific volume
κ		thermal conductivity .

In addition the positions and velocities of the nodes form a field as a function of node indices k and l:

Field	Nodal name in FORTRAN	Definition
$\vec{x}$	R,Z	position as function of k,l
$\vec{u}$	U,W	velocity as function of k,l.

The field equations are

$$\frac{d\mathcal{E}}{dt} = -(p+q)\frac{d\tau}{dt} + \frac{1}{\rho}\nabla\cdot\kappa\nabla\theta \qquad \text{(Eq. 2.2)}$$

$$\theta = \theta(\rho, \varepsilon) \qquad \text{(Eq. 2.3)}$$

$$\kappa = \kappa(\theta) \qquad \text{(Eq. 2.4)}$$

$$q = q(\rho, \Delta\vec{u}, \varepsilon) \qquad \text{(Eq. 2.5)}$$

$$\frac{d\vec{x}}{dt} = \vec{u} \qquad \text{(Eq. 2.6)}$$

$$\rho \frac{d\vec{u}}{dt} = - \nabla(p+q) \qquad \text{(Eq. 2.7)}$$

3. Communications and the Speed and Configuration of a Dataflow Computer

3.1. General issues

Because the least familiar aspect of a dataflow computer is its communications facility, we give a preliminary statement of issues of speed and machine design posed by the burdens that the SIMPLE problem will place on such a facility.

A computational algorithm, such as the FORTRAN program of SIMPLE, defines a flow of data values into and out of arithmetic operations. By analyzing this flow, one can produce a dataflow graph that displays not only the *concurrency* that is allowable within the confines of the algorithm, but also an *abstract pattern of communication*. For the SIMPLE problem, most of the dataflow graph can be modularized onto regions corresponding to the mesh of Fig. 1: one region for each zone, and one for each node.

To perform the computation, resources are required: physical actors must be provided to carry and transform the values that are specified by the dataflow graph. The correspondence between physical actor and role as value carrier is in part subjective, and inescapably so. There is no sure rule for the "right way" to establish the correspondence, although there are criteria by which to exclude many "wrong ways": wrong ways lead to failure (e.g. of performance or of budget). In the light of currently well developed technology, we may start by assigning a physical processor to each nodal and zonal region of the dataflow graph. If each such processor comes with attached memory, then a dataflow computer can consist of a set of processors together with a communications facility that links them.

Affordable communications facilities never offer the full measure of speed, bandwidth, freedom from blocking, and other properties that it would be "nice" to have. Compromise is necessary. The determination of an economic configuration is outside the scope of this work, but to help prepare the ground, we consider the message patterns that are generated by the SIMPLE program. All of the communications facilities under consideration could handle all of these patterns, but different facilities will exhibit different speeds for different patterns. Thus it is helpful to find out what patterns really matter.

The burden placed by a dataflow graph on the communications facility depends on:

.1 the connectivity of the dataflow graph -- how "scrambled" are the needed connections;

.2 the number and accuracy of the field variables to be transmitted.

A given dataflow computer can compute a dataflow graph corresponding to a square mesh of D zones without having to time-share its hardware (as would a sequential computer). Thus D measures the largest mesh that a given dataflow computer can handle in some "fully concurrent" manner. If D is to be increased, then additional hardware must be incorporated into the dataflow computer. In many cases of interest one expects to find $N \gg D$, so that each processor will have to be time-shared among N/D regions. The burden on the communications facility will thus also be influenced by:

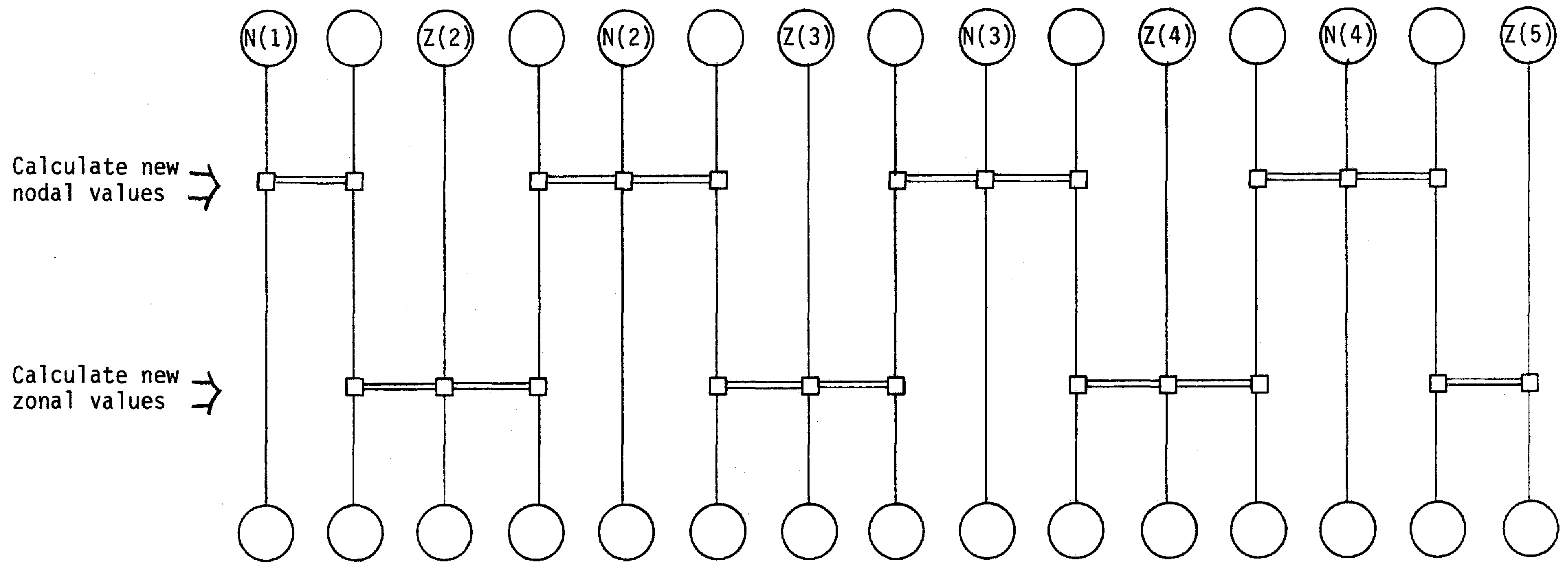
.3 the way in which resources are time-shared over different regions of the dataflow graph.

Item .2 affects only the size of the messages to be transmitted and will not be further considered here. Items .1 and .3 affect the "from-where-to-where" aspect of the communications burden, and we now discuss them further.

3.2. Connectivity in the face of resource sharing

By means of a role diagram, further explained in Appendix A, Figure 2 illustrates the connectivity exhibited by the main cycle of a problem like SIMPLE, but reduced to one space dimension and stripped of heat flow. Figure 2 can be read as a marked graph over which tokens are moved to simulate the occurrence of calculational activities; the top row of circles are viewed as initially marked with tokens. A horizontally connected row of boxes (□═══□═══□) is a calculational activity. The inputs to an activity arrive from above; the outputs depart below -- in other words the "flow of time" is downward. Boxes connected by double bars (□═══□) produce identical copies of the same output value, and thus portray fanout. The figure is thought of as wrapped around a cyclinder, with each bottom circle "wrapped up" to coincide with the circle directly above it, so that a cycle is defined.

The diagram is to be interpreted not just as an abstract flow of values, but as a <u>flow of values carried by physical actors</u>. Each vertical line in Fig. 2 requires a physical resource, like a processor or a buffer, that carries a value from one calculational activity to another. Each horizonatl row likewise specifies a physical requirement -- e.g. for the processing resources needed if the indicated values are to meet and be transformed. The diagram of Fig. 2 looks similar to a dataflow graph because it assumes no constraints due to any scarcity of resources: it assumes that processors and communications links are provided in abundance, at least at the level of detail portrayed. Resource constraints would change the picture; for example, Fig. 3 shows the same values as they would flow under additional constraints imposed by a scarcity of processors such that each processor must handle two adjoining activities.



Note: N(K) = set of values for node K: {P, Q, RHO, E};

Z(K) = set of values for zone K: {X, V}.

Figure 2: Connectivity of simplified hydrodynamics in one space dimension with one processor assigned to each nodal and zonal calculation.

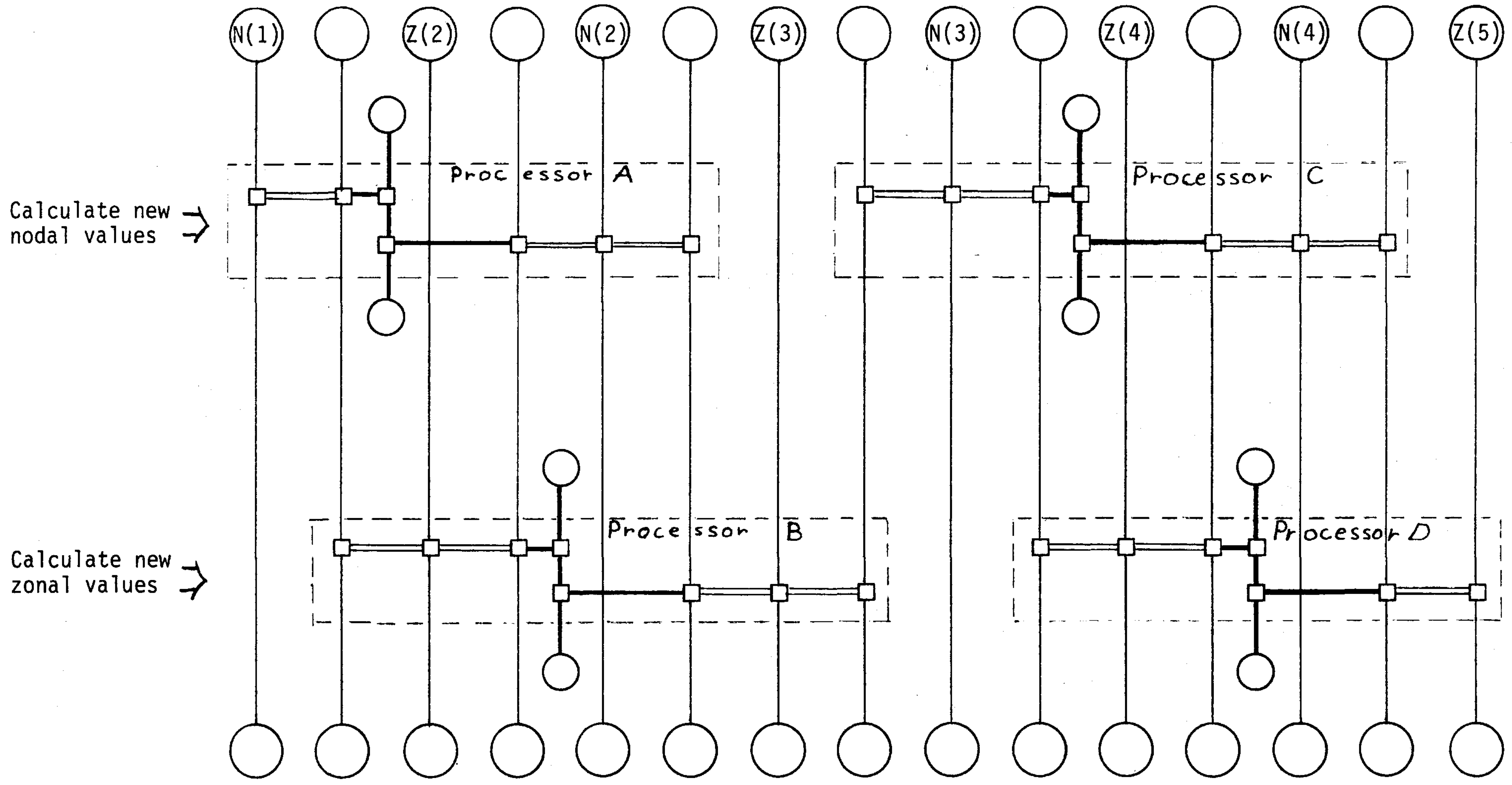


Figure 3: Constraints on concurrency (heavy lines) imposed by sharing of processors.

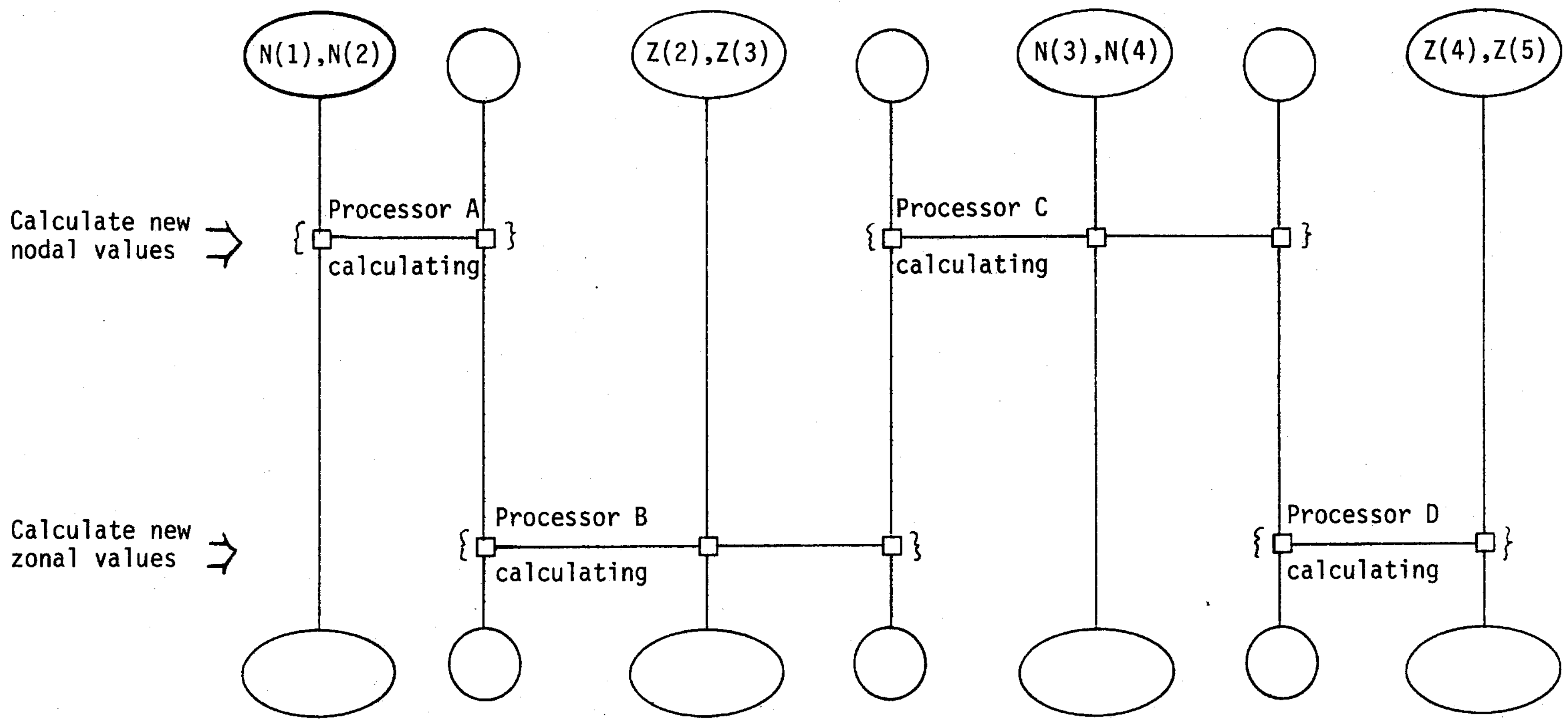


Figure 4: Grosser view of Fig. 3 highlighting connectivity between processors; (compare with Fig. 2).

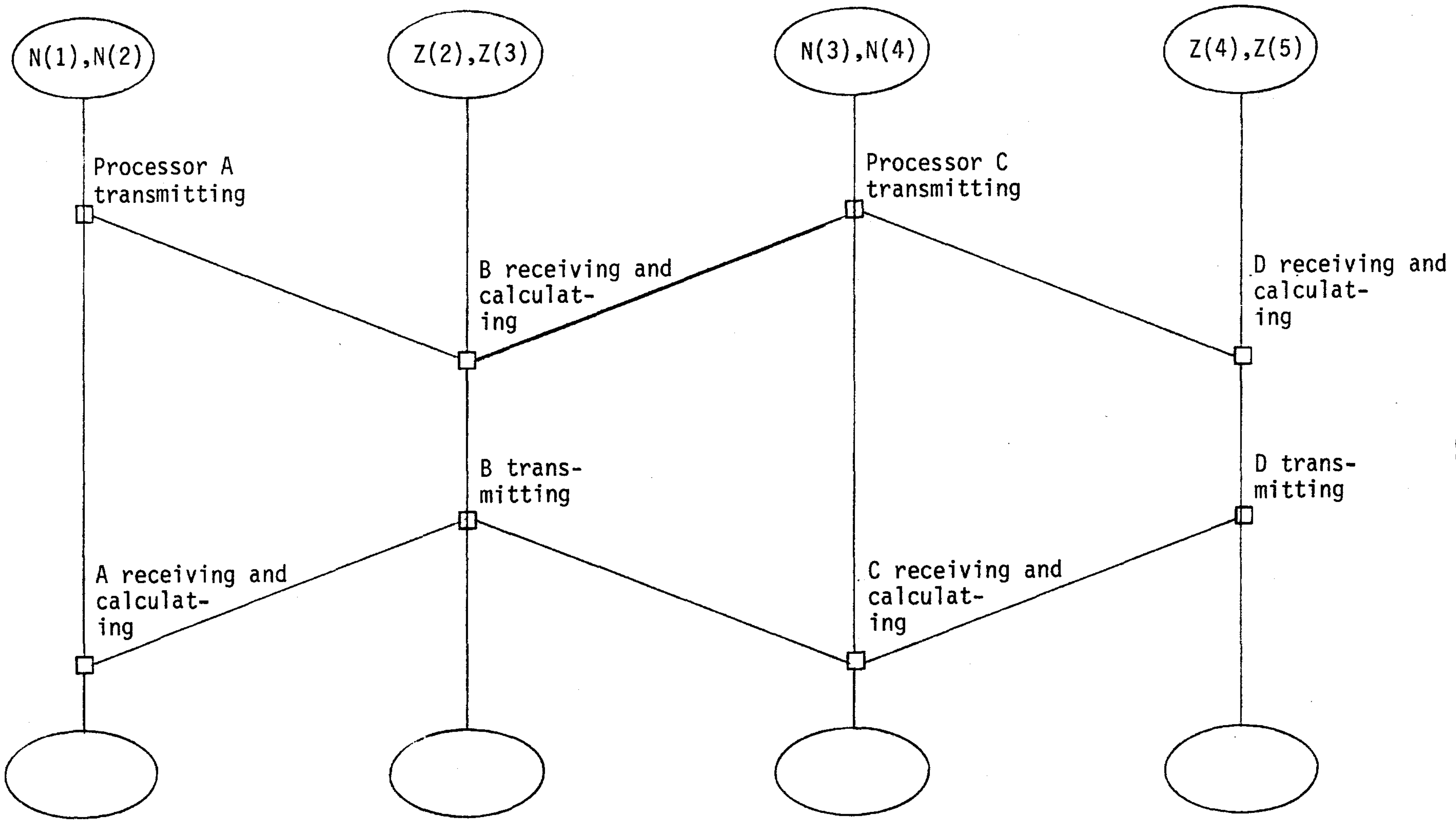


Figure 5: Alternate view of Fig. 4 using the notation of buffered communication.

(See Appendix A, Sec. A.19 for more on the notation.)

The suggested assignment of one processor to one nodal or zonal region of the dataflow graph was in some degree arbitrary. Given a small mesh and many processors, concurrency might be enhanced by assiging more than one processor to each such region. For a mesh large compared to the number of available processors, each processor would have to be assigned a larger piece of the dataflow graph. A question then arises: under this circumstance does simplicity in the connectivity of the dataflow graph imply that simplicity can be maintained in the connectivity of the processors? The answer depends on how a single processor is assigned to cover more than one region. Figure 3 illustrates the principle that such assignment can be made so that the connectivity between processors is no more complex than is the connectivity between nodal and global regions. Figure 4 highlights this connectivity among shared processors; the same connectivity can be maintained when processors are shared over larger regions of the dataflow graph. By use of the abbreviated notation described in Sec. A.19 of Appendix A, Fig. 5 shows the same connectivity as Fig. 4, but with the communications buffers (the unlabeled roles) suppressed. A slanting bar implies: a) that the lower of the activities consumes something produced by the upper activity; and b) that the two activities are linked by an intermediating resource (such as a buffer) that is not explicitly shown.

What can we learn from this example that is more generally applicable? Sharing of processors reduces the size of the communications facility required of a dataflow computer, at the cost of speed. For this example and this manner of assigning processors, the communication pattern, although becoming smaller, preserves its connectivity; be it one or many regions of dataflow graph per processor, each processor communicates only with itself and with its nearest neighbors. In the SIMPLE problem one finds somewhat more complex more connectivity in the dataflow graph. Two points are to be noted in the assignment of processors to pieces of dataflow graph of SIMPLE.

.3. A mesh of N zones can be parcelled out to D processors in such a way that the connectivity among processors preserves any "localness" present in the connectivity among nodal and zonal regions of the dataflow graph.

.4. Other schemes of assigning processors that place additional demands on their connectivity may offer advantages in speed.

Because of item .3 we can learn what connectivity is <u>necessary</u> to D processors of a dataflow computer that is to solve a mesh of N zones, merely by studying the connectivity of the dataflow graph. Because of item .4 we must bear in mind that there will be additional questions of trade-offs between speed, cost, and the connectivity of the communications facility.

3.3. Fitting the Computation to the Minds of the Analysts: Input and Output

Programs and parameters flow into a pattern of computation, and significant features of the computation flow out. In some cases this interaction can be partitioned into a sequence of phases: input, computation, output. However, as the size of the computation increases there is progressively more need to operate interactively, so that the selectivity of what flows out can be increased along with the amount of computation.

Output from a dataflow machine is apt to involve transforming an array, or some feature (such as a contour) extracted from it, into a sequence of characters to be transmitted -- either to a person or to a storage device. Such operations are bandwidth limited and threaten to demand excessive time or buffering or both. As the scale of computation is increased, it becomes necessary to increase the selectivity of feature extraction in near proportion.

One reason that extracting features is challenging is that what is significant sometimes becomes apparent only as the computation unfolds, so that the definer of significance must interact with the computation. Further, significance varies according to the viewer. Because of this "vaporous" quality, one approach is to report out "all the data" from a computation, so that it forms a database that can later be manipulated according to taste. As the scale of computation increases, this approach becomes progressively more demanding, and may become unrealizable.

An alternative approach would be to provide a facility by which multiple viewers of the computation could each construct filters and other "feature extractors" in real time as the computation proceeds. No doubt some users would still build "databases", but they would have the opportunity (and perhaps the necessity) of building more selectively than has been the common practice.

This approach generates requrirements to be met by dataflow hardware and software. The image is of a controllable "funnel" or "tree" that sucks up arrays of field variables as the computation proceeds, discards what is irrelevant, and issues a stream of characters that conveys the features specified by one or another analyst. The "specification of relevant features" could by supplied prior to execution, or could be supplied interactively by the analyst as the computation unfolds.

Such a scheme demands software interfaces that can accept analyst-supplied specifications of the features to be selected. Presumably the structure should accomodate multiple analysts. The hardware requirements are an extension of those already generated by the needs to sum over an array and to convert an array into an output stream for transmission over a single communications line. For example, program-controlled merging of array elements into a stream can provide efficient sorting. Just as they are needed to sum and to report out all the elements of an array, tree structures will be needed to report out selected elements of an array (such as the elements of a contour). However, one expects an advantage from more flexible control of tree connectivity and of tree, nodal and zonal processing than would be needed just to solve the field equations.

4. Modeling the Time to Compute

The prediction of execution time of SIMPLE on a dataflow computer that is not yet fully specified is a complex task which, in this report, can be started but not completed. For this reason we separate a general discussion of what needs to be undertaken from a sketch of initial results.

4.1. Choosing an appropriate form of model

The question of time to compute is a question of what happens when an abstract pattern -- the algorithm of SIMPLE -- meets a configuration of physical resources -- communications lines, switches, buffers, processors, etc. that compose a dataflow computer. The modeling of computation time entails the modeling of the joining of the abstract event of the algorithm with the physical event of the configuration. This calls for a modeling form that straddles abstract (i.e. input-output) relations and physical circumstances. For example, we are forced to observe that anything that is (even a value) must be some place, such as on a communications line, in a buffer, etc. We must learn to see something like a dataflow graph as having, in addition to its implications for abstract values, implications concerning the resources required to support the logical operations on values. As a foundation for this shift in view, we turn to Holt's (1979) concept of the role played by an actor who carries a value. The value is in the domain of mathematics and algorithms; the actor (human or mechanical) is in the domain of space and time.

It would be advantageous to have a gross model with only a few parameters, both to estimate the time for a dataflow computer to solve the SIMPLE problem, and to help in configuring an implementation of a dataflow computer. However, a believable gross model of such a complex situation can be derived only by condensing a model that encompasses sufficient complexity to account, for example, for the effects of pipe-lining and of communications bottlenecks. It thus appears that the modeling form should lend itself to different levels of detail.

The modeling method must encompass the concurrency exhibited by dataflow architecture. This requirement rules out models based on the concept of a system state, and directs toward models based on Petri nets.

The modeling scheme must provide for the modeling of different methods of numerical solution. For example, the implicit formulation of heat flow results in a difference equation, the solution of which is equivalent to the inversion of a certain near-diagonal matrix. The method of inversion used in SIMPLE is that of back-substitution. However, it appears possible to develop an alternative method that would impose far fewer unnecessary sequencing constraints, and would hence better realize the potential advantage of dataflow architecture.

The SIMPLE program uses a global determination of a time step that varies from one cycle to another, but is invariant over the mesh. It appears that in the computation of hydrodynamic shock, there would be a substantial advantage in providing for the local determination of time steps that would vary not only from cycle to cycle, but also from location to location over the mesh. Such methods are used in the calculation of gravitational fields and in relativistic fluid dynamics, as is discussed by Misner, Thorne and Wheeler (1970, Chap. 42). Although this extension of method is outside the scope of our present work, we require that the modeling method encompass time steps as local values derived on an even footing with other field quantities.

These requirements suggest modeling based on the concept of a Petri net. Because of its capacity to join abstract and physical operations, we choose the modeling scheme of Holt (1979) to express the essential logical and physical dependencies. For a discussion of the concepts, the reader is referred to the cited report of Holt. As a "quick and dirty" view of "how to do it", Appendix A describes the modeling conventions.

4.2. The need for speed

Faster computers are desired to allow a finer grained mesh. Consider a given physical domain and a given duration of hydrodynamic interaction. As the mesh is made finer the number of zones, N, increases, and moreover the physical time step achievable in a cycle of computation decreases as $1/\sqrt{N}$. Therefore the time to compute increases as $N^{3/2}$. This dependence applies to a dataflow computer with $D << N$, just as it does to a sequential computer.

To decrease the linear dimension of the zones by a factor of 10, N must increase by a factor of 100, and to maintain a fixed time to compute, given the necessary decrease in physical time step, the speed of the computer must be raised by a factor of 1000.

One should not that the constant of proportionality that relates the allowable physical time step to $1/\sqrt{N}$ depends on the numerical method used, and that the freedom to choose an advantageous method depends on the connectivity of provided by the communications facility of the dataflow computer. Richer (e.g. more than nearest-neighbor) connectivity supports larger time steps, but then richer connectivity slows the computer and requires an investment that could otherwise buy more processors; thus there is a trade off.

4.3. The computational cycle

The SIMPLE computation consists of initialization followed by repeated execution of a main cycle. A cycle consists of computing the velocity and position of each node, and then computing the properties (such as pressure and density) of each zone. The cycle involves times in two senses: a physical time step (e.g DTNPH in SIMPLE); and a time to compute the cycle. Because the initialization is done once and the cycle is repeated many times, the (total) time of computation is nearly independent of the time to initialize the computation, and is essentially the time to compute a cycle multiplied by the number of cycles.

The computational cycle can be partitioned either in terms of the physics or in terms of the concurrency and connectivity that it presents. These two partitionings result in somewhat different pictures. The following is a compromise between the two. We view the cycle as composed of the following phases of activity:

.1. establish boundary values (by means of "ghost" nodes and zones);

.2. calculate velocity and position of interior nodes;

.3. calculate zone variables for interior zones (e.g. pressure, specific energy, artificial viscosity, density) except for temperature;

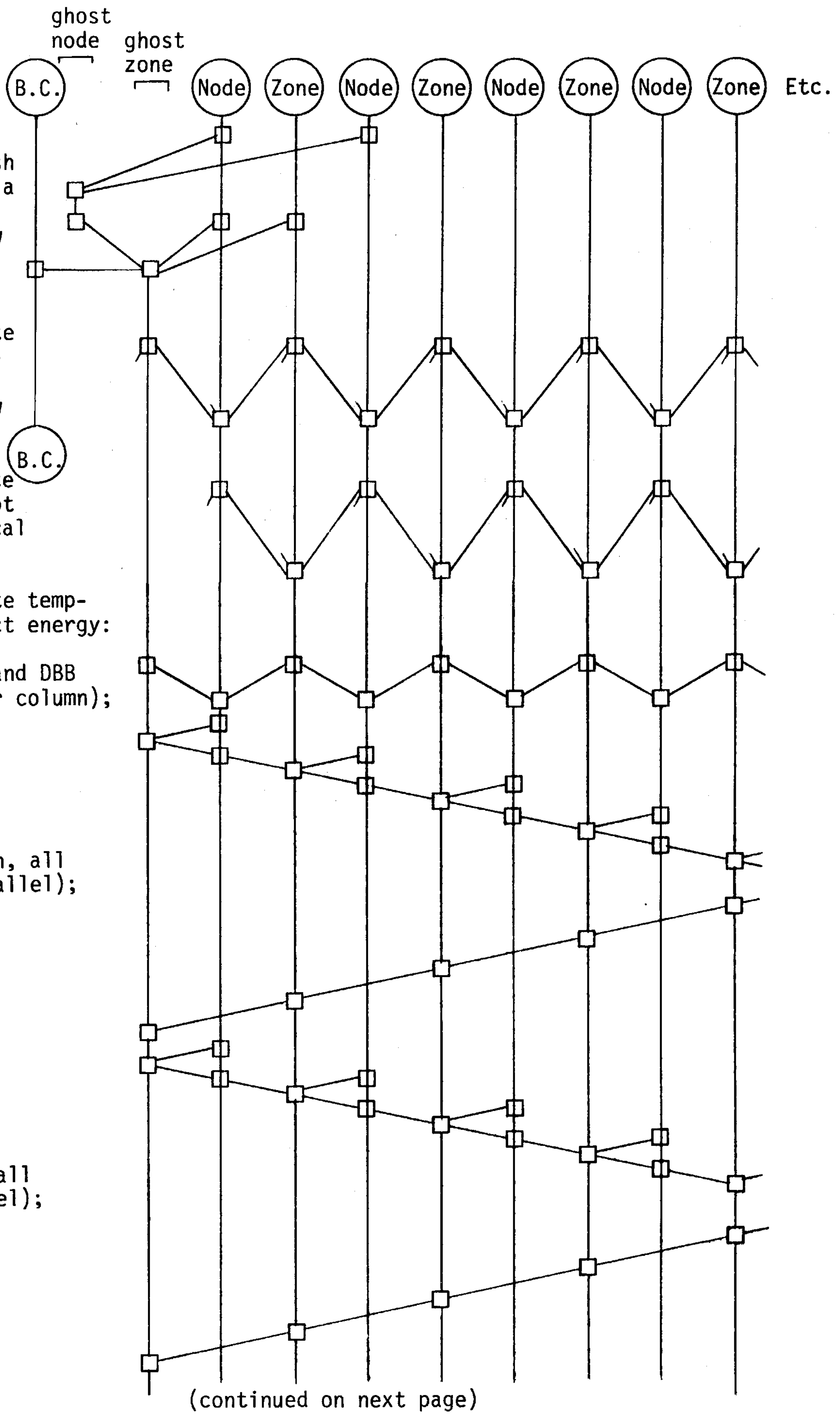
.4. calculate temperature and recalculate energy to include the effect of heat flow;

.5. calculate the time step for the next cycle;

.6. calculate totals: work done on boundary, energy lost, etc.

.7 extract needed output and bring in parameters to control subsequent output, as discussed in Sec. 4.5.

Figure 6 schematically displays the types of connectivity, and hence concurrency, in the flow of data prescribed by SIMPLE over a network of processors, with one processor assigned to each node and each zone of the dataflow graph. Additional processors are assumed to handle the "tree" connectivity of phases 5, 6 and 7. As noted in Sec. 3, if fewer processors are available, they can still be connected with the same connectivity, by assigning each processor a set of contiguous zones, contiguous nodes, or portion of the "tree". If more processors are available, then more than one can be assigned to a given nodal or zonal region of the dataflow graph, with the result that a higher degree of parallelism will be achieved. Some possible assignments of this type are illustrated in Appendix B.



Phase 1: Establish boundary values via ghost nodes and zones (typical row or column).

Phase 2: Calculate velocity and position of interior nodes (typical row or column).

Phase 3: Calculate zonal values except temperature (typical row or column).

Phase 4: Calculate temperature and correct energy:

calculate CBB and DBB (typical row or column);

Z-sweep (typical column, all columns in parallel);

R-sweep (typical row, all rows in parallel);

(continued on next page)

(continued from preceding page)

Phase 5: Calculate next time step and distribute ("tree" connectivity covers all zones):

calculate locally, then take minimum;

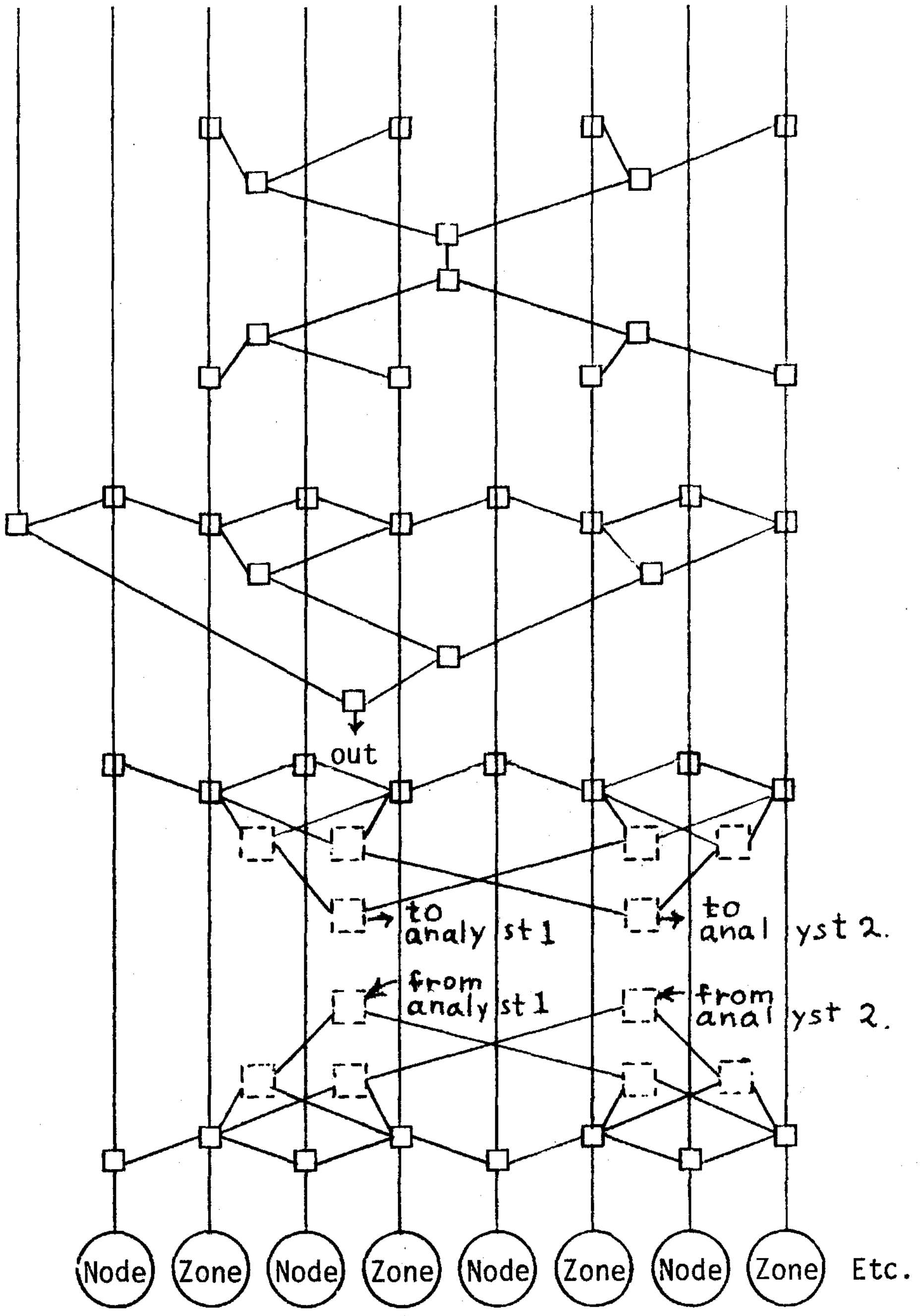
distribute.

Phase 6: Calculate total internal energy and energy exchange across boundary ("tree" connectivity covers all zones; see note a.)

Phase 7: Input/output:

test values (e.g. against thresholds) and extract features (see Note b.)

receive changes in parameters (e.g. thresholds) that control feature extraction. (See Note b and Secs. 3.3 & 4.5.)



Note a: Phase 6 consists of a local calculation, like phase 3, followed by a summing operation. In SIMPLE this phase is distributed throughout the other phases; however, this distribution does not change the character of the demand placed on computational resources.

Note b: The dotted box ( ) will involve sequencing ( ) or not ( ), according to whether messages are or are not concatenated.

Figure 6: Concurrency and connectivity in different phases of the cycle.

4.4. Dependence of time to compute on number of zones and number of processors

Although not attempting quantitative estimates, we discuss how the time to compute varies with the size of the mesh and the number of processors. Each phase of SIMPLE, as shown in Fig. 6, will be considered separately, as different phases exhibit different dependencies. Several areas of uncertainty confront even qualitative estimation; in particular the detailed operation of a communications facility necessary to a dataflow computer bears on the dependence. This operation has not been modeled to date; for this reason we confine our discussion to two limiting cases. The first case leans toward keeping the communications facility local; i.e. communications between nearest neighbors are stressed. The second case posits a general purpose, global communications facility without worrying about its realizability; the intent is to see what contribution to speed such a facility could make if it were available.

4.4.1. Case definitions

Case 1: connectivity restricted to nearest neighbor plus "tree". As case 1 we posit a restricted communications facility. We imagine processors connected like a two dimensional mesh, with a provision for two-way communications between each zonal processor and its neighboring nodal processors. I.e. the processors are divided into two classes, and a given direct communication is always between two members that are in different classes. Fig. 7 illustrates the connectivity. In addition, we posit additional processors and connections to perform such functions as global sums and the taking of maxima. Each zonal processor is imagined to be a twig of a tree. At nodes of the tree there are processors of a third class (the "tree" class) which can operate to

a) accept a flow of values from twig to root, operating by program to select and pass on the largest value, to sum the incoming values and pass on the sum, etc, or

b) accept a value flowing from root to twig, providing either for fanout to all zones or for selective routing to a given zone.

For simplicity we imagine that the mesh of the SIMPLE problem is roughly square, and that the D zonal processors are arranged in a square array. To use the configuration of case 1, we imagine that each zonal processor is assigned about N/D contiguous zones; i.e. each zonal processor operates on a "super"-zone of the mesh, as discussed in Sec. 3. As indicated in Sec. 3., the connectivity between super-zones (and the corresponding super-nodes) will show the same pattern as does Fig. 6. The assignment of pieces of dataflow graph to processors is static, and does not change during execution of the program.

Case 2: "general-purpose" communication. Suppose that the dataflow computer has a communications facility that is ideal in the sense that each processor can send a message to any other, with a rate of flow constrained only by the bandwidth of the processors. We define parameters as follows:

T_e = time for a processor assigned to a node or zone of the dataflow graph of SIMPLE to enter a communication into the communications facility, for forwarding to another processor; and

$T_x(D)$ = time for the communication, under the loading conditions at hand, to travel to its destination.

T_x must increase with D at least logarithmically; in practical terms it will probably grow more or less linearly.

The assignment of processors to portions of the dataflow graph can be made as in case 1, but, as will be discussed below, there is an advantage in speed if processors can be reassigned during execution. In particular, during the Z-sweep of phase 4 it is an advantage to have each zonal processor assigned to a column of zones of the dataflow graph; during the R-sweep it is an advantage to have each zonal processor assigned to a row of zones of the dataflow graph.

4.4.2. Results

Consider the SIMPLE problem for a mesh of N zones, running on a dataflow computer capable of computing a mesh of D zones without time-sharing of hardware. The running time will depend on the time to compute a cycle, as discussed previously. The time to compute a cycle will be a function of N and D. Examination of the connectivity shown in Fig. 6 for various phases of the cycle leads to the results shown in Table 1. In Table 1 the parameters T_1 through T_7 will be different for the two cases, and indeed depend on details of the implementation. However, they do not depend substantially on N or D.

In order to move to a quantitative estimate, one must both estimate the parameters T_1 through T_7 for whatever detailed cases are to be judged, and one must also determine the degree to which pipelining could make the total cycle time less than the sum of the times for the individual phases.

Although the values of the T-parameters may vary between case 1 and case 2, it is to be noted that the dependence on N and D is of the same form for the two computers, except in phase 4, where the configuration of case 2 promises a substantial advantage. This advantage could be obtained as follows. Assume for simplicity that $N = D^2$ and that the mesh is square, so that there is one processor for each row of zones and for each row of nodes, or alternatively, one processor for each column of zones and for each column of nodes. For the Z-sweep assign each processor to a column, so that one processor must operate sequentially along its column. Because of the data dependence of the back-substitution method used, this involves no more computing time than would the assignment of one processor per zone and node. At the completion of the Z-sweep, reassign each processor to a row, in preparation for the R-sweep. In this reassignment each processor must send and receive field variables to and from all the other processors of its class. If the communications facility accepts messages as fast as the processors can stuff them in, then we find that the time to reassign is about as follows:

$$\text{Reassignment time} = D\,T_e + T_x(D) \quad . \qquad \text{(Eq. 4.1)}$$

Table 1, under Phase 4, shows the comparison of dependencies achieved with this capability, versus the simpler facility offered in case 1. (Note that T_4 for case 1 is not the same as T_4 for case 2.) It is to be noted that the advantage of the more general communications facility can be realized only if the facility supports "high bandwidth" in the sense of providing for complete exchange of messages among all processors. This total exchange must actually take place to make the scheme work.

The square-root dependence shown for case 1 comes about because in a square array of processors with processing constrained to be sequential along a column (for example), then only one row of processors is in parallel; the other rows are waiting. As D is increased, the length of the row of processors grows as the square root of D.

	Phase 1: boundary value determination.	Phases 2 & 3: calculate nodes and zones <u>except</u> temperature.	Phase 4: calculate temperature — Case 1: communications restricted to nearest neighbor and "tree".	Phase 4: calculate temperature — Case 2: "general purpose, global" communications facility.	Phase 5: time step.	Phase 6: energy totals.	Phase 7: input/ output.
Numerical method of SIMPLE	$T_1\sqrt{N/D}$	$(T_2+T_3)\frac{N}{D}$	$T_4^{(1)}N/\sqrt{D}$	$T_4^{(2)}N\left(\frac{1}{D}+\frac{T_e}{T_c}+\frac{T_x(D)}{D}\right)$	$T_5\frac{N}{D}\log_2 D$	$T_6\frac{N}{D}\log_2 D$ (note c)	$T_7 N$; but reducible to something approaching $T_7\log_2 N$ through increasing selectivity of feature extraction. See Secs. 3.3 and 4.5.
Change to hypothetical concurrent method for matrix inversion	"	"	(?) $T_4'\frac{N}{D}\log_2 D$		"	"	
Additional change to local determination of time steps	"	"	"		$T_5'\frac{N}{D}$	"	

Notes. a) Communication more general than "tree" + "nearest neighbor", even if available, can be effectively used only in phase 4.

b) The issues of estimating the parameters T_1 through T_6 is discussed in Sec. 4.4.

c) Phases 1 through 6 may overlap, so that, as discussed in Sec. 4.4, the cycle time may be less than their sum; in particular the results of phase 6 are not used in any loop calculation and phase 6 can thus easily be pipelined.

d) The mesh is assumed to be roughly square.

Table 1: Form of dependence of <u>time to compute a cycle</u> on number of zones (N) and number of processors (D).

4.5. Input, Output, and Control Over the Extraction of Features

For the first six phases of Table 1, the time to compute diminishes as the number of processors is increased. But this is not so for phase 7: SIMPLE requires the "wholesale" shipment of arrays to an external storage medium. As discussed in Sec. 3.3, the time to transmit N elements over a single transmission line has a lower bound that is proportional to N, and moreover is independent of how many processors are brought into the dataflow computer. Thus the generation of output threatens to consume a time that could become excessive. This threat can be countered by providing greater selectivity in reporting; i.e. one programs for the reporting only of significant features, and avoids communicating "masses of raw data".

In order to avoid swamping analysts even with present computers, Livermore Laboratory has assembled a powerful facility for computerized extractions of significant features from masses of data. At present the approach is to first compute a relatively "general" database, and then to exercise selectivity in the extraction of features. In order to make efficient use of a dataflow computer, one must shift to a greater emphasis on selectivity in generating the output which will form displays and/or "special purpose" databases. Without bringing selectivity into the generation of output, the linear growth of time to report an array with the number of zones is apt to dominate the computation. Even if it does not, the increase in size in any "general-purpose" database is a serious drawback.

The SIMPLE code offers a small beginning in this direction in the option in the EDIT subroutine by which one can eliminate the reporting of nodes and zones that show less than a specified degree of motion. More is doubtless done in other programs to provide selective reporting, but still more must be done as the scale of computations is increased. As a specific example along these lines, an analyst could specify that the value of say pressure be reported out for any given zone <u>only</u> if the pressure had changed by more than ten percent since the last report for that zone. Thresholds (e.g. the "ten percent") might be varied during execution.

If selectivity of reporting is made to increase in near proportion to the number of zones, then input and output can be handled with a structure for which Phase 7 of Figure 6 serves as a point of departure. As discussed in Sec. 3.3, however, more trees and more flexible control over them would be of advantage. The goal of selectivity would be to keep the formation of output from overwhelming the analyst and from taking too long. Through increasing selectivity with the number of zones, one can keep the growth rate of the time to form the output from growing as fast as the number of zones; one might hope to contain it to a logarithmic dependence.

Further discussion is outside the scope of this work, but would be appropriate for a future project.

5. Translation of SIMPLE from FORTRAN into VAL

5.1. The balancing of objectives

In developing a code in any language, the following desires are balanced:

.1. Express the algorithm as clearly as possible; and

.2. Make good use of computing resources.

In producing VAL code for a dataflow computer whose hardware is not yet fully specified, it would also be desirable to illuminate constraints on concurrency, and in particular to:

.3. Organize the code so as to make clear which aspects of SIMPLE place which demands on hardware speed and connectivity; and

.4. Extend the SIMPLE problem by sketching more of the input and display functions, because these functions are essential to any actual problem of the SIMPLE type and place demands on both language and hardware not made by other phases of the problem.

In addition, since we are translating from FORTRAN, it would be desirable to:

.5. Make VAL code that can easily be compared with the given FORTRAN code.

These desires conflict in various ways, and any VAL code will reflect a balance between them. In support of items .1 and .3 we group variables into bunches (such as START) in a way that will either decrease efficiency or place extra burdens on compilation. The decrease in efficiency would take the form of sending a longer message where a shorter one would suffice;

concurrency at the level of detail shown in Fig. 6 would not be affected.

In support of items .2 and .3 we have sacrificed item .5 to the extent of introducing new variables (STRESS, GX, GV) that are tensors defined in each zone, in order to demonstrate that the connectivity demanded by SIMPLE in computing the acceleration of each node is only nearest-neighbor, in contrast to the first impression given by lines 580 throug 593 of SIMPLE (1979). Appendix B illustrates demands placed on hardware by various parts of the SIMPLE problem, as expressed in VAL.

In support of .4 we have indicated possible extensions of the VAL language that seem to be needed to help with the extraction of significant features from an array, and with input and output in general; these are:

a. the stream type of value for input and output;

b. the addition of concatenate to the list of forall operations, so that a stream can be formed quickly from a sparse array;

c. the addition of an asymmetric merge operation on arrays to help in communicating a sparce pattern of change to an array; the effect is that one of the two arrays to be merged supplies default values which are overridden by non-empty elements of the other array.

d. a form of forall eval max that extracts the lowest index at which the maximum value of an array of reals is found, in addition to the maximum value itself.

In support of item .5 we use the names of variables as given in the FORTRAN code except where different structures are introduced.

In connection with item .1, it is to be noted that the algorithm of

SIMPLE evolved over decades in a process that was influenced by often conflicting needs for single-step accuracy, stability, and economy; for this reason the algorithm will not be found to show a simple structure, no matter how it is displayed.

The FORTRAN code, including comments, runs some 1749 lines, and a complete translation into VAL would be of roughly the same size. Because the SIMPLE code in FORTRAN is always undergoing minor revisions, as is the VAL language, it seems beside the point to carry through details of translation that duplicate the form of translations already made. We rely on Hirshman (1978) and Woodruff (1979) to demonstrate that many FORTRAN passages can be translated efficiently into VAL; some of thes passages are referred to in what follows. Rather than duplicate their work, we present a more detailed code of the main module of the VAL program for SIMPLE, as a framework in which to view passages that deal with specific acitivities of computation. In this framework we highlight the issues that were encountered in a detailed review of the entire SIMPLE program, focusing on areas, notably input and output, that require further development of the VAL language. Our intent is both to show how the present edition of VAL is sufficient to translate most of the FORTRAN, and to show clearly certain extensions of VAL that appear necessary for a complete translation, including the extension of SIMPLE to provide for the extraction of significant features from arrays.

5.2. Samples of VAL code

5.2.1. Overall form of the VAL translation of the SIMPLE code

As discussed by Ackerman and Dennis (1979) a VAL program consists of a collection of external function modules, each of which may contain internal function modules. One internal module cannot invoke another. We present the VAL code for SIMPLE as a main external function module called SIMPLE_VAL, along with an external function JES which is a table look-up used by two functions internal to SIMPLE_VAL; in addition some external routines presumed to be in a system library are used, such as sine, cosine, and square root. The bulk of the code will be the function modules internal to SIMPLE_VAL.

Each external function module consists of:

header,

type definitions,

external function declarations (e.g. for library supplied utilities)

internal function definitions, and

body.

In the code that follows there will be gaps, indicated by comments, such as passages that can be filled in from the work of Hirshman (1978). Comments will also indicate where a possible extension of the VAL language has been invoked to overcome one or another obstacle of the type discussed in Sec. 5.1.

The program will consist of the external functions

```
SIMPLE_VAL
JES_VAL
SIN
COS
SQRT  %square root,
```

and might well be augmented by system utilities to indicate running time, etc.

Because certain features of SIMPLE_VAL are understandable only in the context of JES_VAL, we present JES_VAL first.

5.2.2. JES_VAL

The FORTRAN code of SIMPLE contains a table look-up subroutine named JES. In SIMPLE_VAL this look-up is used by two internal functions: ENERGY_HYDRO and ENERGY_HEAT. Because it is called by two internal functions, we construct the VAL translation of JES as a function external to SIMPLE_VAL.

JES operates on numbers and not arrays; it can be applied fully concurrently be each zonal processor to the elements of a given zone.

An issue in translating is that the FORTRAN version of JES uses many GOTO statements, and these statements are not supported under the more structured philosophy of VAL. Thus the JES code must be re-expressed in an IF-THEN-ELSE form. In arriving at the code displayed below, it was very helpful to first flow chart the FORTRAN CODE. Another issue is that in FORTRAN, JES is employed not by calling "JES", but by calling one or another of the entry points IES1 and IES2; these will correspond to the parameter ENTER in JES_VAL, our VAL equivalent of JES: ENTER = 1 corresponds to IES1; ENTER = 2 corresponds to IES2.

Partly because it uses a method of successive approximations, SIMPLE employs JES several times in the calculation of energy for a single zone. JES (for ENTER=2) returns energy or (for ENTER=1) pressure as a function of temperature (TARG1) and density (RARG1), by means of a table look-up. The table is organized as a two dimensional array of rectangular regions on the (temperature, density)-plane, with a region specified by a pair of integers NT and NR. The returned value is supplied by a procedure that has several steps:

- Search for and find the NT, NR for the region that contains the "point" (TARG1, RARG1);

- Per line 1353, statement 5310 of SIMPLE (1979), evaluate a function of NT and NR to obtain an integer M as index to an array of sets of coefficients -- e.g. AES[M], etc. The set of coefficients for a found M will be used to interpolate.

- Obtain the value to be returned by means of a quadratic interpolation function, using the set of coefficients AES[M], etc.

The running time of SIMPLE (at least for a sequential machine) is significantly reduced by saving NT, NR, and M as NTSV[N], NRSV[N], and MSV[N] for use as trial starting values for the search in the next invocation of JES. In the FORTRAN code NT (along with NR and M) is saved separately according to which of the two entry points (corresponding to ENTER = 1 or ENTER = 2) is invoked. Thus NT is saved in a two-element array, with one element for each possible entry point. We refer to the six saved numbers collectively as SV_REC, where SV_REC is a structure of type SV_REC_type, defined by:

```
type SV_REC_type = record[NT, NR, M: array[integer]] %.
```

The structure which we have called SV_REC saved from a given zone supplies trial values for the next invocation of JES, which may be for the same zone, or for a different, usually neighboring zone, as the sequential processor steps from zone to zone. The facilitation of the search is still likely when a shift is made to a neighboring zone, because conditions change little from a zone to its neighbors. The speed advantage accrues because the sequential processor usually last invoked JES either for the same zone or for a neighboring zone. When the last invocation was for a far-away zone, then SV_REC is no help; this does not affect the answer produced by JES, but does extend the time to find the answer.

Now we turn to the issue of translation for a dataflow computer. Suppose, as suggested in Sec. 3, a dataflow computer has D zonal processors, each assigned to cover a "super-zone" composed of (about) N/D contiguous zones. When $N \gg D$ a given zonal processor will step sequentially from zone to zone in a "raster scan" over its N/D assigned zones, just as the sequential computer is specified by the SIMPLE code to scan all N zones. There are three options:

a. Omit the use of SV_REC, and accept a slower look-up (noting that because many look-ups will be done concurrently, the speed is not so important as it was in the FORTRAN code).

b. Create an array of SV_REC's, with one SV_REC for each zone. This option maintains the speed, but as the cost of storing a factor of N/D more SV_REC's than are really needed.

c. Cause each zonal processor to carry one SV_REC along as it steps through its N/D zones.

Option a) is easiest to implement, but is hardly an example of translating power. Option c) is both the most efficient and the most demanding, and is coded in Sec. 5.2.3, where it shows up in initializing SV prior to entering the main loop, and in Sec. 5.2.4 where it is discussed under ENERGY_HYDRO.

The VAL function module follows:

```
function JES_VAL(ENTER: integer; TARG1, RARG1: real; SV_REC: SV_REC_type
                 returns real, SV_REC_type)
type SV_REC_type = record[NT, NR, M: array[integer]]
let % The closing "in" is the          the last line of JES_VAL.
% Set up constants for table; these are provided in the FORTRAN code by
% subroutine SETUP acting via COMMON; we incorporate much of the equivalent
% of SETUP here.
IZES, ITES, IRES: array[integer] := [1: ...], ... ;
TES, RES, AES, ... , PES: array[real] := [1: ...], ... ; % End of set-up part.
EXTT1, EXTR1: real := 1;
N: integer := ENTER; % Change of name to conform to FORTRAN code
NT, NR: integer := SV_REC.NT[N], SV_REC.NR[N];
EXTT2: real := EXTT1 * TARG1;
EXTT, TARG: real, FLAG, NT1: integer :=
if TES[NT] > TARG1 then
   if NT <= ITES[N]then EXTT2 / TES[NT], TES[NT], Ø, NT
   else for N1: integer := NT-1
   do if TES[N1] > TARG1 then
          if N1 >  IES[N] then iter N1 := N1-1 enditer
          else EXTT2 / TES[N1], TES[N1], 1,N1 endif
      else EXTT1, TARG1, 1, N1 endif
   endfor
   endif
else if TES[NT+1] > TARG1 then EXTT1, TARG1, Ø, NT
    else if NT+2 = ITES[N+1] then EXTT2 / TES[NT+1], TES[NT+1], Ø, NT
        else for N1: integer := NT-1
           do if TES[N1+1] > TARG1 then EXTT1, TARG1, 1, N1
```

```
                else if N1+2=ITES[N+1]then EXTT2 / TES[N1+1], TES[N1+1], 1, N1
                     else iter N1 := N1+1 enditer endif
                endif
             endfor
          endif
     endif
endif
EXTR2: real := EXTR1 * RARG1;
EXTR, RARG: real, FLAG2, NR1: integer:=
if FLAG=0 then
   if RES[NR] > RARG1 then
      if NR > IRES[N] then for N1: integer := NR-1
         do if RES[NR] > RARG1 then
               if NR > IRES[N] then iter N1 := N1-1 enditer
               else EXTR2 / RES[N1], RES[N1], 1, N1 endif
            else EXTR1, RARG1, 1, N1 endif
         endfor
      else EXTR2 / RES[NR], RES[NR], Ø, NR endif
   else if RES[NR+1] > RARG1 then EXTR1, RARG1, Ø, N1
        else if NR+2=IRES[N+1]then EXTR2 / RES[NR+1], RES[NR+1], Ø, NR
             else for N1: integer := NR+1
                do if RES[N1+1] > RARG1 then EXTR1, RARG1, 1, N1
                   else if N1+3 > IRES[N+1] then EXTR2/RES[N1+1], RES[N1+1], 1, N1
                        else iter N1 := N1+1 enditer endif
                   endif
                endfor
             endif
        endif
   endif
else if RES[NR] < RARG1 then for N1: integer := NR
        do if RES[N1+1] > RARG1 then EXTR1, RARG1, 1, N1
           else if N1+3 > IRES[N+1]then EXTR2/RES[N1+1], RES[N1+1], 1, N1
                else iter N1 := N1+1 enditer endif
           endif
        endfor
     else for N1: integer := NR
        do if RES[N1] > RARG1 then
              if N1 > IRES[N]then iter N1 := N1-1 enditer
              else EXTR2/RES[N1], RES[N1], 1, N1 endif
           else EXTR1, RARG1, 1, N1 endif
        endfor
     endif
endif;
M: integer := if FLAG2=0 then SV_REC.M
              else IZES[N]+(ITES[N+1]-ITES[N]-1)*(NR1-IRES[N]+NT1-ITES[N]) endif;
SV_REC1: SV_REC_type :=
   if FLAG2=0 then SV_REC
   else SV_REC replace[NT: SV_REC.NT[N: NT1]; NR:SV_REC.NR[N:NR1];
   M: SV_REC.M[N:M]]           endif;
FUNC: real  := AES[M] + RARG * (BES[M] + RARG * DES[M])
             + TARG * (CES[M] + RARG * (FES[M] + RARG * GES[M])
             + TARG * (EES[M] + RARG * (HES[M] + RARG * PES[M])));
FUNC1 : real := if ENTER=1 then FUNC * EXTT * EXTR
                else FUNC * EXTT endif
in %closes "let" on line 2 of JES_VAL
FUNC1, SV_REC1 endlet endfun % End of function JES_VAL
```

5.2.3. SIMPLE_VAL

SIMPLE_VAL is the main module -- i.e. the overall framework -- for the VAL code translation of SIMPLE. Because the functions internal to this module correspond to roughly 25 pages of FORTRAN code, the section of internal function definitions is abbreviated to a list of headers, and a discussion of salient features of these modules will be found in Sec. 5.2.4. The code that follows is a detailed statement of the overall structure of the VAL translation of SIMPLE.

```
% Header:
% Note presumed language extension to "stream" type for input and output.
function SIMPLE_VAL(INPUT_A:  start-type;  INPUT_B: stream
   [correction_type] returns stream[out_phys_type],
   stream[out_cycle_type], stream[out_edit_type],
   stream out_condition_type )

%type definitions:
type vector = record[R,Z: real];
type zonal = array[array[real]];
type zone_tensor = array[array[record[E,W: vector]]];
type nodal = array[array[vector]];
type node_scalar = array[array[real]];
type start_type = record[DTNPH, TFLR, EDDT, PØ, EØ, RHOØ, DTMIN,
   DTMAX, TMAX, CØF, C1F, GAM: real; BC: record[U, D, L, R: integer];
   LIM: record[KN, KX, LN, LX, DS: integer]; NCP: integer];
```

```
% As shorthand we shall write "STATE" and "state_type" to refer to
% a list of the variables that define the state of the computation:
% state_type = list[DTNPH, DTN, TNUP, ENCG, EDTIME, EDDT: real; NYCL:
%    integer; P, Q, RHOJ, E, S: zonal; X, V: nodal; GX: zone_tensor;
%    DTMIN, DTMAX, TMAX, CØF, C1F, GAM, EDDT, TFLR: real; NCP: integer]
 type out_phys_type = "state_type";
 type out_cycle_type = record[NYCL: integer; DTNPH, TE, ENC, SKE, HN, WN,
                              ENCG: real; DTEN, DTC2: record[DT: real;
                              K, L: integer]];
 type out_edit_type = "state_type";
 type out_condition_type = stream; % language extension
 type correction_type = stream;

 type lim_type = record[KN, KX, LN, LX, DS: integer]; % 4 fields correspond
 % to FORTRAN code KMN, KMX, LMN, LMX; DS describes implementation for
 % the implementation-dependent use of JES_VAL shown in ENERGY_HYDRO.

 type SV_REC_type = record[NT, NR, M: array[integer]];
 % SV_REC discussed in Sec. 5.2.2 in connection with JES_VAL.

 type SV_type = array[array[SV_REC_type]];  % Because of our choice of
 % option c) of Sec. 5.2.2, the array SV of type SV_type will have
 % dimensions of LIM.DS by LIM.DS, where  LIM.DS squared is D, the
 % number of zonal processors of the dataflow computer.  If option b)
 % were used, then LIM.DS would not have to appear in the program, and
 % the array SV would have N (number of zones in mesh) elements instead
 % of D elements.
```

```
        % external function declarations:
        external JES_VAL(ENTER: integer; TARG1, RARG1: real; SV_REC:
           SV_REC_type returns real, SV_REC_type)
        external sin(DUMMY: real returns real)
        external cos(DUMMY: real returns real)
        external sqrt(DUMMY: real returns real) % square root.


% The bodies of the internal function definitions are omitted here; the
% headers are listed for all internal functions of SIMPLE_VAL:

%     INITIALIZE(START: start_type returns "state_type")

%     EDIT(STATE returns edit_type)

%     BOUNDARY_PROJECT(P, Q, RHOJ: zonal; X: nodal; GX: zone_tensor; LIM:
%                      lim_type returns zonal, zonal, zonal, zone_tensor)

%     VELOCITY(V: nodal; P, Q, RHOJ: zonal; GX: zone_tensor; DTN: real;
%              LIM: lim_type returns nodal)

%     POSITION(X,V: nodal; DTNPH: real; LIM: lim_type returns nodal)

%     HWORK(X, V: nodal; P, Q: zonal; DTNPH: real; LIM: lim_type returns real)

%     ZONE_GEOM(X, V: nodal; MASS, S: zonal; LIM: lim_type returns
%               zonal, zonal, zonal, zonal, zone_tensor, zone_tensor)

%     ENERGY_HYDRO(E, P, AJ, RHO, DVOL, MASS: zonal; GX, GV: zone_tensor;
                   SV: SV_type; DTNPH, CØF, C1F, GAM, DTMAX: real; LIM:
                   lim_type returns zonal, zonal, zonal, zonal, SV_type)
```

```
%     HYDRO_TOTAL(V: nodal; MASS, E: zonal; LIM: lim_type returns real, real, real)

%     ENERGY_HEAT(E, RHO, AJ, TEMP, MASS: zonal; X: nodal; SV: SV_type;
                 DTNPH, TFLR: real; LIM: lim_type returns zonal, zonal, zonal,
                 node_scalar, node_ scalar, SV_type)

%     HEAT_TOTAL(E, TEMP, MASS: zonal; CBB, DBB: node_scalar; DTNPH, HN: real;
%               LIM: lim_type returns real, real)

%     TIME_STEP(TSO, YE: zonal; X: nodal; DTNPH, DTMAX, CØF, C1F, GAM: real;
%              LIM: lim_type returns real, real, real, real)

%     PHYS_REPORT("STATE": "state_type" returns "state-type")

%     CYCLE_REPORT(YE, TSO: zonal; NYCL: integer; TNUP, DTNPH, TE, ENC,
%                 SKE, HN, WN, ENCG: real; LIM: lim_type returns
%                 out_cycle_type)

%     MODIFY("STATE": "state_type"; DUMMY: correction_type returns
%           "state-type")
% body of SIMPLE_VAL
% The gross plan of the body is
% for STATE: state_type:= INITIALIZE(first(INPUT_A));
%      OUT_PUT: stream:= null
% do if (condition) then OUT_PUT
%     else iter STATE:= main_cyle(STATE) enditer
%     endif
%     endfor
% In the detailed presentation that follows we split "STATE" into
% its fields (as given in the section of type definitions), and split
% "main_cycle" according to the phases illustrated in Figure 6:
for START: start_type:= first(INPUT_A); % read input stream
    STATE: "state_type":= INITIALIZE(START);
    OUT_PHYS: stream[out_phys_type]:= null;
    OUT_CYCLE: stream[out_cycle_type]:= null;
    OUT_EDIT: stream[out_edit_type]:= EDIT(STATE);
    CORRECTION: stream := INPUT_B;
    HN, WN: real := 0.;
    % Set up temporary variables, other than those covered in STATE,
    % needed for main cycle:
    AJ, DVOL, TEMP, TSO, YE: zonal := array_fill(LIM.KN + 1, LIM.KX,
       array_fill(LIM.LN + 1, LIM.LX, 0.));
    GV: zone_tensor := array_fill(LIM.KN + 1, LIM.KX,
       array_fill(LIM.LN + 1, LIM.LX, record[E, W: record[R, Z: 0.]]);
    CBB, DBB: nodal := array_fill(LIM.KN, LIM.KX, array_fill
      (LIM.LN, LIM.LX, record[R,Z: 0.] ));
    DTEN, DTC2, SKE, ENH, TE, ENC: real :=0.
LIM: lim_type := START.LIM;

% Set up array of SV_REC's to conform to option c) of Sec. 5.2.2.
% Let DS be the greatest integer such that DS*DS  = D, where D is the
% number of zonal processors, as discussed in Sec. 3.
SV: SV_type :=
   let DS: integer := LIM.DS % implementation-dependent parameter.
   in array_fill(1, DS, array_fill(1, DS, record NT: array_fill(1, 2, 0);
   NR: array_fill(1, 2, 0); M: array_fill(1, 2, 0); EXTR: 0.)) endlet;


do if DTNPH < DTMIN | TNUP > TMAX then
      let OUT_CONDITION: stream :=
          if DTNPH < DTMIN then "DT_STOP" || NYCL ||TNUP ||DTNPH || DTMIN
          else "STOP TMAX" ||NYCL ||TNUP ||TMAX  endif
      in OUT_PHYS, OUT_CYCLE, OUT_EDIT, OUT_CONDITION endlet
   else iter
% Phase 1 of cycle (see Fig. 6 for description of phases):
P, Q, RHOJ, GX := BOUNDARY_PROJECT (P,Q, RHOJ, X, GX, LIM);

% Phase 2 of cycle:
V := VELOCITY(V, P, Q, RHOJ, GX, DTN, LIM); % vector velocity
X := POSITION(X, V, DTNPH, LIM);  % vector position

% "WN" part of Phase 6:
WN := HWORK(X, V, P, Q, DTNPH, LIM) + WN;

% Phase 3 of cycle:
RHO, AJ, DVOL, S, GX, GV := ZONE_GEOM(X, V, MASS, S, LIM);
E, P, Q, TEMP, TSO, SV := ENERGY_HYDRO(E, P, AJ, RHO, DVOL, MASS,
                              GX, GV, SV, DTNPH, CØF, C1F, GAM, DTMAX, LIM);
```

```
% Hydro part of phase 6:
SKE, ENH, TE := HYDRO_TOTAL(V, MASS, E, LIM);

% Phase 4 of cycle:
E, RHOJ, YE, CBB, DBB, SV := ENERGY_HEAT(E, RHO, AJ, TEMP, MASS, X, SV,
                                          DTNPH, TFLR, LIM);

% Heat part of phase 6:
ENC, HN := HEAT_TOTAL(E, TEMP, MASS, CBB, DBB, DTNPH, HN, LIM);

% Phase 5 of cycle:
DTN, DTNPH, DTC2, DTEN := TIME_STEP(TSO, YE, X, DTNPH, DTMAX,
                                    CØF, C1F, GAM, LIM);

% Phase 7 of cycle (output and corrective input):
OUT_PHYS, EDTIME :=
   if TNUP < EDTIME then OUT_PHYS, EDTIME
   else OUT_PHYS || PHYS_REPORT(STATE), EDTIME + EDDT endif;
NYCL := NYCL + 1;
OUT_CYCLE :=
   if MOD(NYCL, NCP)~= 0 then OUT_CYCLE
   else OUT_CYCLE ||CYCLE_REPORT(NYCL, TNUP, DTNPH, YE, TSO,
      TE, ENC, SKE, HN, WN, ENCG) % lines 766-773 of FORTRAN
   endif
STATE, CORRECTION :=
   if CORRECTION = null then STATE, CORRECTION
   else MODIFY(STATE, first(CORRECTION)), rest(CORRECTION)
   endif
```

```
% An alternative approach to output would be to extract significant
% features.  For example, we illustrate a report of pressure for only
% those elements of the array P that have changed by at least 10
% percent since they were last reported.  We assume an array P_LAST
% as an iteration variable to carry the "last reported" value of P:
P_LAST, OUT_PHYS_SELECTIVE :=
   if TNUP < EDTIME then nil %language extension for iteration variables
   else let COND: array[array[boolean]] :=
      forall K in [LIM.KN + 1, LIM.KX], L in [LIM.LN + 1, LIM.LX]
      construct   ABS((P[K,L] - P_LAST[K,L])/MAX(EPS, P_LAST[K,L])) < .1 endall
        in forall K in [LIM.KN + 1, LIM.KX], L in [LIM.LN + 1, LIM.LX]
           construct if COND then P_LAST[K,L]
                     else P[K,L] endif endall, OUT_PHYS_SELECTIVE ||
           forall K in[LIM.KN + 1, LIM.KX], L in [LIM.LN + 1, LIM.LX]
           eval concatenate %language extension
              if COND then null
              else record[P: P[K,L]; K: K; L: L] endif endall
        endlet
   endif
% end of example of feature extraction
enditer
endfor
endfun  % SIMPLE_VAL
```

5.2.4. Discussion of functions internal to SIMPLE_VAL

INITIALIZE includes code like the modules GENBC and GENPOS of Hirshman (1978), along with code of the form, say for pressure,

```
% P: zonal :=
array_fill(LIM.KN + 1, LIM.KX , array_fill(LIM.LN + 1, LIM.LX, START.PØ)).
```

EDIT is straightforward to translate, except for one demand which it places on the language: one needs to extract not only the maximum element of an array (as can be done with forall eval max) but also the K,L coordinates at which the maximum is found. Efficient support of this need requires hardware and language attention.

BOUNDARY_PROJECT includes the module GEOMETRY of Hirshman, the filling of P, Q, and RHOJ arrays (where RHOJ[K,L] = RHO[K,L] * AJ[K,L]), and the calculation of GX for boundary zones. The calculation of GX for interior zones is done in ZONE_GEOM, and is discussed in Appendix B.

VELOCITY: see Appendix B, where connectivity of the flow of data is discussed.

POSITION is like Hirshman's module HYDRO; see also Appendix B.

HWORK is essentially Hirshman's module of the same name.

ZONE_GEOM produces AJ and S like the module GENAREA of Hirshman, and also produces GX and GV by the algorithm discussed in Appendix B.

ENERGY_HYDRO contains parts like NEWE and NEWQ of Hirshman. However, NEWQ can be recast to use GX and GV in place of X and V, with the result that the calculation for a given zone draws only on values of that zone; i.e. no node-to-zone communication is required for the computation of Q when

GX and GV are made available from ZONE_GEOM.

Subroutine TEMPCAL of the FORTRAN code can be translated readily into a function module internal to ENERGY_HYDRO. Both via TEMPCAL and directly, ENERGY_HYDRO calls the external function module JES_VAL to compute pressure (from JES_VAL(1, TEMP, RHO, SV_REC)) and energy (from JES_VAL(2, TEMP, RHO, SV_REC)). The value SV_REC supplied to JES_VAL is in effect a hint where to start searching in a table; the value supplied does not affect the numerical results produced by JES_VAL, but it does affect the time to execute JES_VAL.

If option b) os Sec. 5.2.2 were selected, coding into VAL would be easier because there the array SV would have N elements and be of the same shape as P, RHO, etc. For that option a typical use of JES_VAL would be the production of a trial pressure P1, as in:

```
.1
      P1, SV: zonal :=
         forall K in [ LIM.KN+1, LIM,KX], L in [LIM.LN+1, LIM.LX ] construct
         JES_VAL(1, TEMP[K,L], RHO[K,L], SV[K,L]) endall; %.
```

Instead of using option b), we have chosen option c) as an example of the kind of demand on expressive power that occurs in tailoring an algorithm to an implementation. As discussed in Sec. 5.2.2 option c) saves storage by taking SV to be an array of only D (= number of zonal processors) elements; this can be much smaller than the N-element array used in option b). To express the N-element array P1 as a function of a D-element SV, it appears necessary to first create a partitioned array equivalent to P1, with a block of this partitioned array corresponding to an element of SV.

The N interior zones of the mesh constitute a two-dimensional array of (LIM.KX - LIM.KN) by (LIM.LX - LIM.LN) elements. For simplicity we assume that both of these dimensions are exactly divisible by LIM.DS, where D = $(\text{LIM.DS})^2$ is the number of zonal processors used, and we assume a physical configuration of a square array of LIM.DS by LIM.DS zonal processors.

Each zonal processor is to be assigned a rectangular "super-zone" of the mesh, consisting of KS by LS contiguous zones, where

.2

KS = (LIM.KX-LIM.KN)/LIM.DS

and

LS = (LIM.LX-LIM.LN)/LIM.DS .

In place of .1 one expressed an N-element P1 in terms of a D-element SV, where one elemnt of SV corresponds not to one element of P1, but rather to a block of KS by LS elements of P1. Let P_BLOCK be a <u>partitioned</u> array equivalent to P1; that is, while P1 is a 2-dimensional array of reals, P_BLOCK is an array of LIM.DS by LIM.DS "little" arrays, each with KS by LS real elements, so that P_BLOCK must be a 4-dimensional array of reals. Option c) demands that:

- computation proceed in each of the D blocks of P_BLOCK concurrently, and
- within a given block, computation proceed in a raster scan sequentially.

The correspondence between P1 and PBLOCK is:

.3

P1[K1*KS + KØ,L1*LS+LØ] = P_BLOCK[K1,L1,KØ,LØ] .

In other words, K1,L1 tell which block, and KØ,LØ tell which element within the block. It follows that (with the VAL convention for downward rounding of integer division) the [K,L]element of P1 is given by

.4

P1[K,L] = P_BLOCK[K/KS, L/LS, MOD(K,KS), MOD(L,LS)] .

The VAL code for producing P1 and SV in accordance with option c) follows:

```
P1: zonal, SV: SV_type :=
let P_BLOCK: array[array[array[array[real]]]] , SV1: SV_type :=
forall K1 in [1, LIM.DS], L1 in [1, LIM.DS]
       KS: integer := (LIM.KX-LIM.KN)/LIM.DS; % Assume exactly divisible
       LS: integer := (LIM.LX-LIM.LN)/LIM.DS; % "
construct % P_BLOCK[K1,L1] is itself a 2-dimensional array.
   for BLOCK: array[array[real]]:= array_empty[array[real] ; % Element of P_BLOCK.
       SV_REC1: SV_REC_type := SV[K1,L1];
       KØ: integer := 1
   do if KØ > KS then BLOCK, SV_REC1
      else iter BLOCK, SV_REC1 :=
              let BCOL: array[real], SV_REC2: SV_REC_type :=
                 for BCOL1: array[real]:= array_empty[real];
                     SV_REC3 : SV_REC_type := SV_REC1;
                     LØ: integer := 1
                 do if LØ > LS then  BCOL1, SV_REC3
                    else iter BCOL1, SV_REC3 :=
                            let P_EL: real, SV_REC4: SV_REC_type :=
                               JES_VAL(1, TEMP[K1*KS+KØ, L1*LS+LØ],
                               RHO[K1*KS+KØ, L1*LS+LØ], SV_REC3)
                            in BCOL1[LØ: P_EL], SV_REC4 endlet;
                            LØ := LØ + 1
                         enditer
                    endif
                 endfor
              in BLOCK KØ: BCOL , SV_REC2 endlet;
              KØ := KØ + 1;
           enditer
      endif
   endfor
endall
in % P1: zonal, SV: SV_type :=
   forall K in [LIM.KN+1,LIM.KX], L in [LIM.LN+1, LIM.LX] construct
   P_BLOCK[K/KS, L/LS, MOD(K,KS), MOD(L,LS)] , SV1
endlet % Completes production of P1 and SV.
```

Because of the explicit reference to LIM.DS, a parameter of implementation, this example gives a glimpse of the type of expression needed when a programmer assists in compilation. It is generally recognized that hardware can be used more effectively if the programmer tailors the program to it. In simple cases one hopes that the algorithm will not have to be changed to effect such tailoring, but we have just seen a case in which the algorithm (though not its numerical result) did change. To facilitate compilation of the whole SIMPLE code, one might well express all the arrays in blocked (i.e. partitioned) form for internal computation. If this were done then the conversion to 2-dimensional form would not be done as part of the above example, but would be deferred to the generation of output, as in the module PHYS_REPORT of SIMPLE_VAL.

HYDRO_TOTAL, like HWORK, is straightforward, being essentially the execise of the construct forall-eval-plus.

ENERGY_HEAT is the main bottleneck in the SIMPLE problem, because of the sequencing constraints due to the back-substitution method chosen for solving for heat flow. The sequencing constraints are illustrated in Appendix B, Fig. B.1. The constraints are in the "R-sweep" and "Z-sweep" portions of subroutine CONDUCT of the FORTRAN code of SIMPLE. This code steps from one element of an array to another, using results of a previous element to calculate a next element.

Subroutine CONDUCT saves TEMP as TS in line 1586, and then restores TEMP to TS in line 1673, so that after the execution of CONDUCT, TEMP is unchanged; what is calculated is really a temporary variable which we call TEMP1 in the code below. Its use is not to get a new TEMP, but rather to help in adjusting E to account for heat flow. The FORTRAN code partially inializes arrays A and B outside of the sweeps; we incorporate this initialization into the sweeps. The VAL arrays CBB and DBB are like those of the FORTRAN code, but re-indexed to clarify the connectivity actually required (see note be following the VAL code below). The production of TEMP1 in the VAL code for ENERGY_HEAT would then appear inside a LET construct as follows:-

```
% Z-sweep  (per line 1612 of the FORTRAN code of subroutine CONDUCT)
TEMP1: zonal := let TEMP2: zonal :=    % Z-sweep calculates TEMP2
forall K in [LIM.KN + 1, LIM.KX] construct
let A, B: array[real]:= % range over L
    for L: integer := LIM.LN +1;
        ACOL, BCOL: array[real]:= array_fill(LIM.LN, LIM.LX, 0.), TEMP[K]
    do if L > LIM.LX then ACOL, BCOL
       else let DUM1: real := SIG[K,L] + CBB[K,L] + CBB[K,L-1] *  (1 - ACOL[L-1])
            in iter ACOL, BCOL := ACOL[L: CBB[K,L]/ DUM1], BCOL[L: SIG[K,L] *
                    TEMP[K,L] + CBB[K,L-1]* B[K,L-1] /DUM1];
                    L := L+1
    enditer endlet endif endfor
```

```
% ... ALPHA, BETA FORWARD
in for L: integer := LIM.LX; TCOL: array[real]:= TEMP[K]
   do if L < LIM.LN + 1 then TCOL
      else iter TCOL := TCOL[L: A[L] * TCOL[L+1] + B[L]];  L := L-1 enditer
endif endfor endlet endall  % end of Z sweep; returns TEMP2
in % Feed TEMP2 through R-sweep to produce TEMP1:
% R sweep
let A, B: array[array[real]] :=
   for K: integer := LIM.KN + 1; A2D, B2D: array[array[real]] :=
       array_fill(LIM.KN, LIM.KX, array_fill(LIM.LN, LIM.LX, 0.)), TEMP2
   do if K > LIM.KX then A2D, B2D
      else let ACOL, BCOL: array[real] :=
               forall L in [LIM.LN + 1, LIM.LX] DUM1: real := SIG[K,L]
                 + DBB[K,L] + DBB[K-1,L] * (1- A2D[K-1,L])
               construct DBB[K,L] / DUM1, SIG[K,L] * TEMP2[K,L] +
                         DBB[K-1,L] * B2D[K-1,L] / DUM1
               endall
           in iter A2D, B2D := A2D[K: ACOL], B2D[K, BCOL];
   enditer endlet endif endfor
% ALPHA, BETA FORWARD SWEEP
in for K: integer := LIM.KX; T2D: array[array[real]] := TEMP2
   do if K < LIM.KN + 1 then T2D
      else iter T2D := T2D[K:
                forall L in [LIM.LN + 1, LIM.LX]
                construct A[K,L] * T2D[K+1,L] + B[K,L]
                endall]; K := K-1
enditer endif endfor endlet endlet % Returns TEMP1
```

Notes:

a. In VAL the syntax for operating on a two-dimensional array with a forall construct over one index and a for-iter over the other index is different according to which index is subjected to which construct. For this reason the Z-sweep and the R-sweep, which look much the same in FORTRAN, look different in VAL.

b. The FORTRAN code uses an awkward convention in indexing CBB and DBB, with the result that there appears to be more coupling of array elements than is in fact the case; to clarify this we write CBB[K,L] in place of what in the FORTRAN code would be written CBB[K-1,L]; similarly we write DBB[K,L]in place of DBB[K,L-1].

c. In FORTRAN only one edge of the array A is initialized prior to the loop; in VAL it was convenient to initialize the whole array. The VAL code re-initializes A in the R-sweep. This is permissible because although the A array is operated on in the Z-sweep, the only column that matters (i.e. LIM.KN) is not changed in the Z-sweep.

HEAT_TOTAL uses forall eval plus.

TIME_STEP combines Hirshman's module TINCR with the calculation of DTEN, which in the FORTRAN is done in subroutine CONDUCT. Calculation of KC, LC, KEN, and LEN is not done in TIME_STEP, but is deferred to CYCLE_REPORT.

PHYS_REPORT is similar to EDIT.

CYCLE_REPORT is straightforward except for needing the coordinates of an array where a maximum or minimum value is found, as was the case with EDIT.

MODIFY is an augmentation of SIMPLE to allow for real-time interaction with an analyst; e.g. MODIFY is to provide for receiving a change in say DTMAX, or even for receiving an entire "STATE", as would be needed to restart the computation after an analytic "catastrophe".

6. Conclusions and Possible Next Steps

6.1. Speed, input-output, and expression of the abstract algorithm

As shown in Table 1, except for outputting results, the application of D processors configured as a dataflow computer can reduce the execution time of the SIMPLE code by a factor of at least $D^{\frac{1}{2}}$. The sequencing constraints that limit improvement to this factor occur in the calculation of heat flow, as illustrated in Fig. 6. These constraints stem from the method chosen in the SIMPLE code for the inversion of a tri-diagonal matrix: back-substitution. It would appear feasible to find or develop a method with weaker sequencing constraints. If this were done, then all phases of the program, except output, would execute in times that decrease at least as D/log D with increasing D.

As discussed in Sec. 4.5, the outputting of results called for in the SIMPLE code amounts to a "dump" of raw data. There is a minimum time for such a dump that grows with the size of the mesh and is independent of D. As discussed in Sec. 4.5 and illustrated at the end of Sec. 5.2.3, it appears essential to pre-process the data so as to extract significant features. If this is done, then output need not be a bottleneck.

The VAL language is demonstrated as satisfactory for the expression of the SIMPLE problem as an abstract algorithm, provided that certain extensions are made in it. These extensions are listed in Sec. 5.1 and their use is shown in Secs. 5.2.3 and 5.2.4. The need for additional extensions to promote efficiency of execution is discussed below.

6.2. Implications of the spatio-temporal structure of the algorithm

Following Holt (1979) we have analyzed the SIMPLE problem as given in an abstract algorithm expressed first in FORTRAN and then translated into VAL. The algorithm expressed in either language is called 'abstract' when it is viewed as independent of physical arrangements in space and time for its execution. Our analysis of the SIMPLE algorithm in terms of role diagrams reveals spatial and temporal structure which will have to be found in any and all implementations. For example, by tracing through the algorithm for possible references to computational variables we discover the existence of algorithm-defined times when some number n of such variables must be co-maintained. This in turn implies that in any implementation of the algorithm there will have to be available, for some period, a space large enough to hold n values. (As the algorithm is to be executed by electronic circuits, this number n places a lower bound on the physical space which the algorithm can occupy.) To be more specific, E[J,K], P[J,K], Q[J,K], etc. meet in a zone and phase shown in Fig. 6 and in a relational sense define a time and location.

As a second example, we discover in Fig. 6 that for any instruction of the main loop there are times -- i.e. phases -- when a given instruction may be executed and times when it certainly will not be. In other words one can determine prior to execution and independent of implementation that in any given phase a certain large majority of the instructions of the main loop will not be called. This property can be used both to guide compilation and also to guide the design of hardware for a dataflow computer: it suggests a programmable instruction cell that can make ready first one instruction and then another, much like a sequential processor.

Finally the discussion of Sec. 3 and Figs. 3, 4 and 6 show that only a few of the myriad possible patterns of communication are actually needed for a set of processing resources to execute the SIMPLE problem. In configuring a dataflow computer there are many possible alternatives for the arrangement of processing units, instruction cells, packet memory, and communications resources. Different arrangements offer different advantages for different problem classes, and place different demands on compilation. As discussed in Sec. 3, any hardware arrangement will reflect compromises which will detract from the execution of some classes of problems. Prior to large-scale investment, these relations between physical arrangement and problem class need to be examined in connection with various sample problems.

6.3. The balance between programming ease and efficient use of hardware

As a first step in exploring relations between hardware and problem class, VAL was employed to help express a problem in hydrodynamics in support of two anticipated tasks, relative to a dataflow computer that is not yet fully specified:

.1. the design task of choosing a physical arrangement of hardware resources suitable to the SIMPLE problem; and

.2. the compilation task of mapping the coded problem into machine instructions appropriate to a given physical arrangement of resources.

Both tasks concern the mapping of a problem onto physical resources. The mapping is done in two steps: coding in a source language (VAL); followed by compilation which maps the source language into machine instructions. Historically a source language has been intended for the expression of a problem as an abstract algorithm -- 'abstract' meaning that the algorithm was not tied to a particular physical arrangement of resources. But note:

.3. To achieve <u>efficient</u> use of resources a programmer must allow for at least some features of implementation (e.g. "multiply" takes longer than "add").

.4. If the physical arrangement changes too much, a given source language become inappropriate.

Indeed concurrently operability of resources contributed to the need to express concurrency in the problem, and hence to the need for VAL; i.e. VAL is superior to FORTRAN in expressing concurrency. A source language is shaped in part by assumptions concerning the physical arrangement of computational resources. FORTRAN was designed to facilitate a two-step mapping of a problem to machine instructions. In step one FORTRAN is used to map the problem essentially into instructions for a machine that is an <u>idealized</u> sequential computer -- idealized for instance in that it is imagined to have a random-access memory so big as not to be a constraining factor. In step two the FORTRAN code is compiled into machine code for an actual machine that departs in limited ways from the idealization -- e.g. by using a "small" random-access memory backed up by secondary storage.

As FORTRAN corresponds to an idealized sequential computer, VAL presently corresponds to an idealized dataflow computer -- e.g. a dataflow computer imagined to have so many instruction cells that the number poses no constraint on how a problem might be executed. Note that:

.5. A program like SIMPLE is a major task for programmers who can afford to learn the salient features of implementation;

.6. The program is expected to run many hours per execution, and to be executed many times on a machine that costs enough to justify a large investment in efficient execution;

.7. The program is written to answer questions of physics that are progressively better answered as larger mesh sizes become executable in a day's run; the need for answers to these questions justifies a large investment in speed of execution.

Whatever hardware design is chosen, the resources of a dataflow computer will be more complex than those of a sequential computer, and less susceptible to fully automated resource allocation. Within the dataflow context, the balance between ease of programming and efficiency weighs more toward the demand for efficiency. For problems of the SIMPLE type it appears unwise to force a separation between source-language programming and resource allocation. Some current languages -- e.g. PL/1 -- provide facilities for the control of resources; however these facilities are added ad hoc to a language that conceptually is inhospitable to the expression of physical arrangements in time and space. Because VAL encompasses the expression of concurrency, it offers at least a chance of extension to cover the control of resources in a more systematic way. The discussion of ENERGY_HYDRO in Sec. 5.2.4 illustrates a related issue, the adaptation of the algorithm to a particular physical arrangement.

6.4. Extending VAL to support resource allocation

We have seen that the SIMPLE problem has spatio-temporal structure that is germane to physical design, and for a given design, germane to the allocation of physical resources to execute the problem. Presently, a VAL program is thought of as having a "meaning" only to the extent that it defines a dataflow graph at the descriptive level of machine instructions. At this level of description the dataflow graph of SIMPLE is an enormous lacework, with something on the order of a thousand computational events per zone, times thousands of zones. If a compiler works only from a dataflow graph at this level of detail, is it reasonsable to imagine that it could efficiently distribute all the instructions throughout the "space-time" of the computational resources?

One might hope for some future "genious" to design such a compiler, but there is another approach:

.1. Recognize that compilation will in fact use higher-level and/or auxiliary descriptions of the problem in allocating resources; and

.2. Extend the programmer's task and his power of expression -- VAL -- to express properties of the problem that can greatly reduce the burden of compilation -- properties such as those expressed in the role diagram of Fig. 6.

In this approach the programmer would be supported in structuring the problem in a way that eases compilation for a given machine organization. This requires that the programmer be more explicit in guiding the "when" and "where" of program execution. It might be objected that such guidance depends too much on the details of a particular implementation, but this is not necessarily so. There is a middle ground, where the programmer would formally express the information now conveyed by Fig. 6. The "where" implied by a "zone" of Fig. 6 is not directly a "machine location", but rather a relational location inherent in the SIMPLE algorithm. In that algorithm E[J,K], P[J,K], Q[J,K], etc. meet many times, and in a relational sense meetings define "times and locations" -- e.g. Zone[K,L] of Fig. 6. In essence we see the programmer as calling the compiler's "attention" to grosser regions of a dataflow graph than appear at a machine-instruction level of description. The compiler would thus block out the assignment of gross regions to resources in a first phase, and then subsequently deal with further details. To pursue this course additional effort is needed to:

.3. Bring under control the expression of the space-time aspect of an algorithm at different levels of detail, so as to guide the algorithm toward a particular machine organization;

.4. Show what changes would be needed for VAL to express such aspects;

.5. Evaluate the advantage of expressing SIMPLE and other examples in this way with respect to:

  a. what suggestions are offered for the organization of the resources of a dataflow computer; and

  b. how to distribute the burden of computing a problem over a given organization of resources.

REFERENCES

Ackerman, W. B. and J. B. Dennis (1979) "VAL -- A Value-Oriented Algorithmic Language; Preliminary Reference Manual" Massachusetts Institute of Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, Massachusetts 02139 (March 22).

Crowley, W. P., C. P. Hendrickson and T. E. Rudy (1978), "The SIMPLE Code", Lawrence Livermore Laboratory Report UCID 17715, February 1.

Dennis, J. B. (1978) "Data Flow Computer Architecture", Computation Structures Group Memo 160, Laboratory for Computer Science, Massachusetts Institute of Technology (May)

Hirshman, D. S. (1978) "SIMPLE, A Lawrence Livermeor Laboratories Program Translated into Data Flow Language", Massachusetts Institute of Technology, Laboratory for Computer Science, Computation Structures Group (May 18)

Holt, A. W. (1979) "Roles and Activities, A System for Describing Systems" (Incomplete draft) Boston University, Academic Computing Center, 111 Cummington Street, Boston, Mass. 02215

Misner, C. W., K. S. Thorne and J. S. Wheeler (1970) "Gravitation", W. H. Freeman and Co., San Francisco.

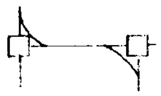
Morse, P. M. and H. Feshbach (1953) "Methods of Theoretical Physics" McGraw-Hill Book Co, New York.

SIMPLE (1979): FORTRAN code of Lawrence Livermore Laboratory, Edition of February 12 as provided by John Woodruff. (Reproduced in App. C.)

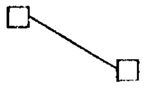
Woodruff, J. P. (1978) VAL code for one-dimensional hydrodynamics (Edition of December 4).

Appendix A: Interpreting Role Diagrams

SECTION DIRECTORY

A.17.  Saving an old value.

A.18. Operations (+, -, etc.)

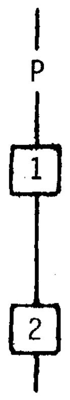
A.19.  Buffered communication.

Appendix A: Interpreting Role Diagrams

Throughout the report we have used role diagrams, invented by A. W. Holt (1979) to show the flow of values carried by physical actors. The notation presented here allows us to distinguish participations of actors in activities according to whether they are coincident, concurrent, alternative, or sequenced.

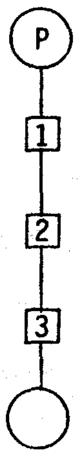
The interpretation of role diagrams differs from that of dataflow graphs in that the former is based on this attitude: anything that is (even a value) must be someplace. Hence the flow of a value is a flow of effect over physical actors. A role diagram can be partitioned into strips; each strip is a locality in system space, and thus a place where some actor is resident.

A.1: A vertical line is read downward as the advance of a role player (i.e. an actor) from one state to another through a sequence of activities. A state is drawn as a vertical line segment; an activity is drawn as a box. Here we show a role player "carrier of the value PRESSURE" proceeding through activity 1, followed by activity 2.



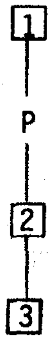
A.2: The vertical line can be thought of as marked by a token. The position of the token shows the state of the role player. The token for pressure carries an inscription which states the value of the pressure.

A.3: Circles at the top and bottom of a vertical line denote the same location of a circuit. In other words the figure



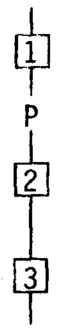
denotes a cyclic progression through activity 1, activity 2, activity 3, back to activity 1, and so on.

A.4: If a role P is initialized in activity 1 and terminated in activity 3 we draw the following.

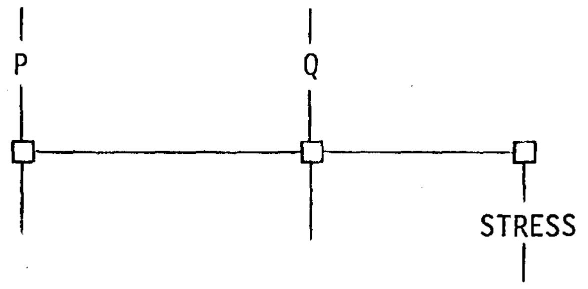


Note that the initiation of a role (shown in activity 1) requires that a physical actor be on hand to play the role.

A.5: In contrast to A.4, a fragment of a longer chain is drawn

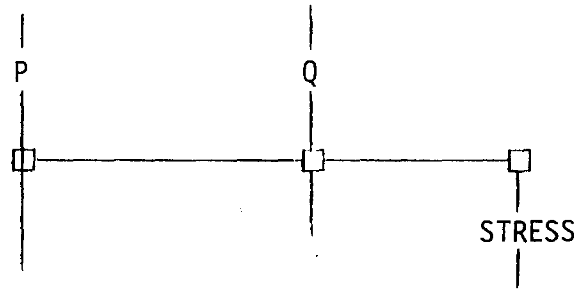


A.6: When several roles participate in a common activity their coincident participation is denoted by horizontal links.

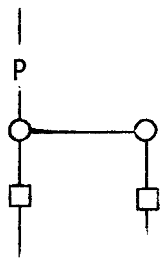


As shown, P and Q must coincidently be present at the creation of STRESS. The horizontal line of boxes converts inputs (above) to outputs (below).

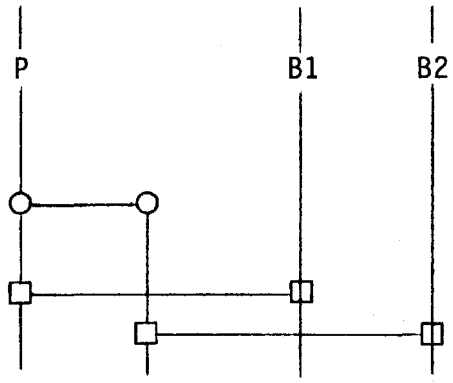
A.7: The diagram A.6 indicates that P and Q change values as a consequence of taking part in the creation of STRESS. If we wish to indicate no change of value of P, we draw



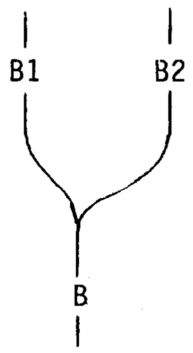
A.8: A role can branch into alternative states, shown as



A.9: In case of a branch, the choice of path can be resolved by interaction with other value-carrying roles. Suppose that exactly one of B1 or B2 will be present, and will resolve the choice for P; then A.8 could be filled out as

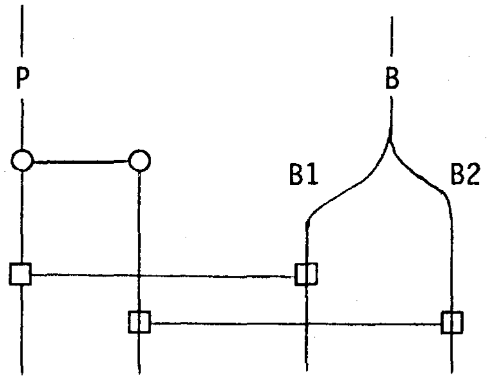


A.10: In drawing a diagram with two alternative states, such as B1 and B2 in A.9, it may be convenient to pull the two lines into one:

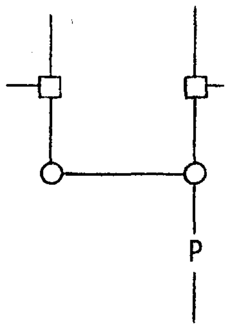


This pulling together is not an "objective" fact of the "system", but rather a matter decided by the drawer of the diagram. He decides to view the distinction formerly borne by the separation of the lines as "encoded" into an attribute of a token that travels on the joined line.

<u>A.11:</u> If the person who draws the diagram has <u>encoded</u> B1 and B2, as in A.10, then in drawing A.9 he would have to "<u>decode</u>" them -- i.e. to reporduce separated lines, one for each of the encoded alternatives. In this case A.9 would be drawn with a fork:



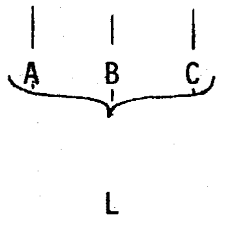
<u>A.12:</u> Two activities can be alternatives to the production of a single state, in which case two states of a role can <u>merge</u>.



A.12 can be compared with A.9. Lines joined by branches and merges of a role form a state component of a Petri net.

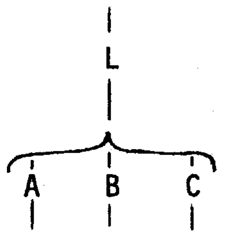
<u>A.13:</u> For convenience of presentation one may wish to bundle several roles together and picture them as a single "cable", as in an image of cabling together of different "wires". We illustrate this by roles A, B and C which are "cabled" into a compound strand called L . In other words,

$$L = \{A,B,C\}.$$

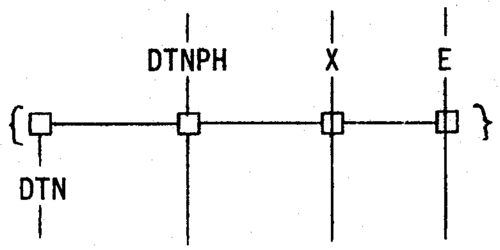


Unlike encoded alternatives (see A.10) all the roles of a bundle can be concurrently played

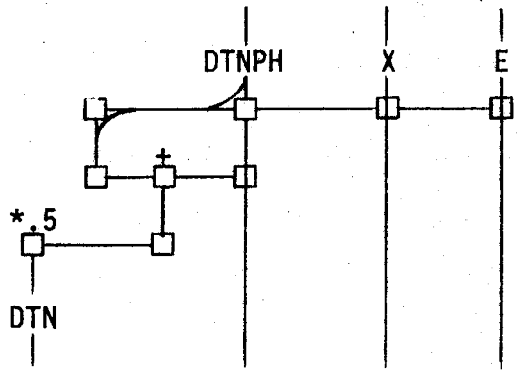
A.14: Unbundling corresponding to the undoing of A.13 is drawn as follows.



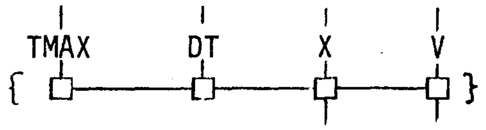
A.15: Brackets around a row indicate that the row is compressed from a more detailed diagram shown elsewhere; for example the figure



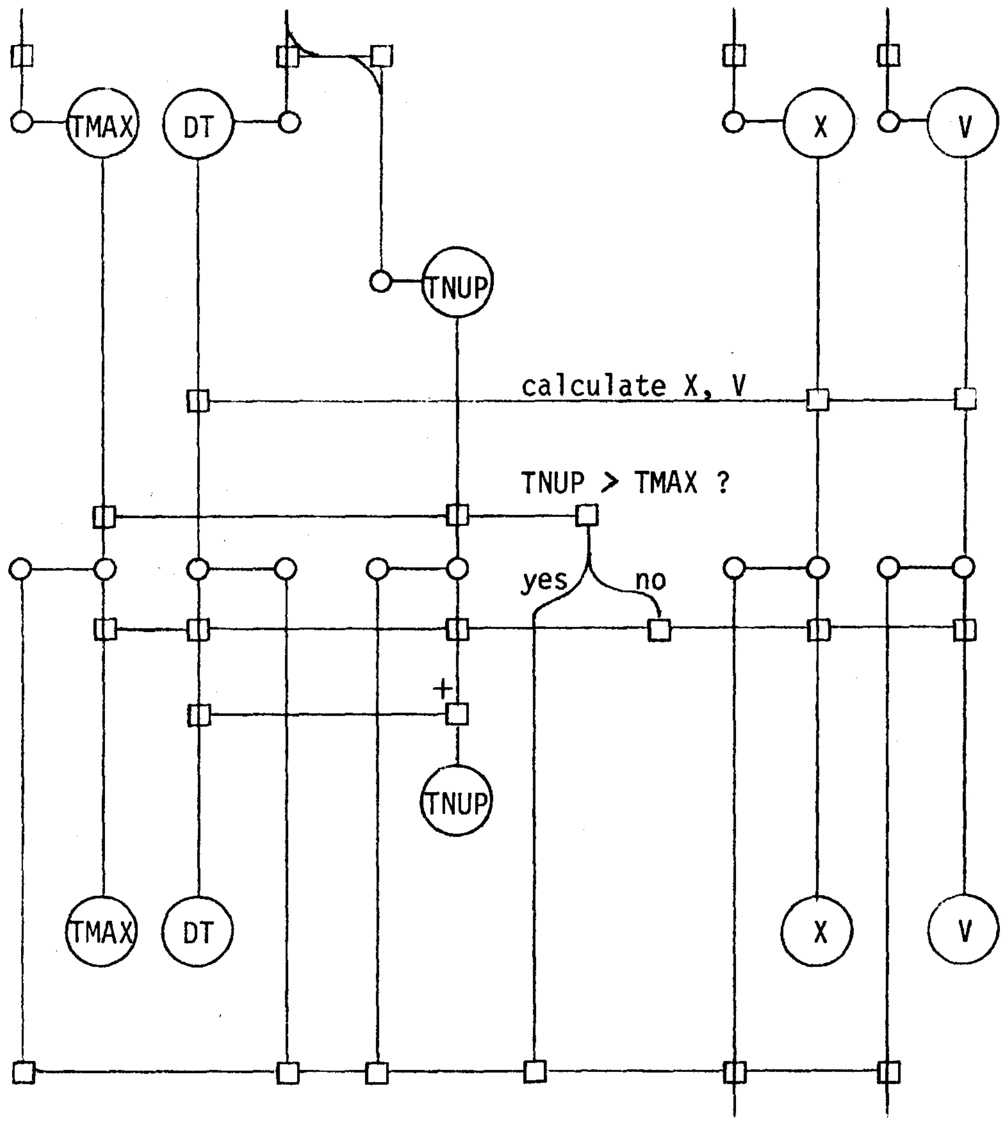
is compressed from



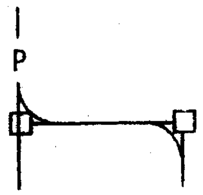
A.15.1: The outputs of a bracketed row can be produced by an internal loop, containing internal variables. TNUP is such a variable in the following diagram, where



is compressed from

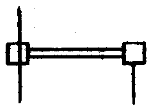


A.16: The following illustrates *fanout*.

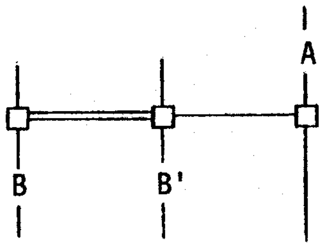


(This notation was used in A.15.1.)

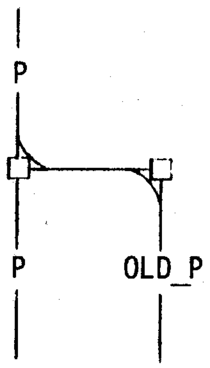
A.16.1: Fanout can also be shown as follows.



A.16.2: We link two boxes by a double bar to assert identity of output values; the following asserts that after the occurrence of the activity, B and B' carry copies of the same value; the figure does not assert anything about the relation between inputs, nor about the relation between inputs and outputs.

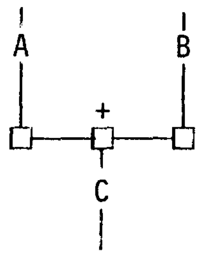


A.17: The following illustrates the saving of the value of P as OLD_P, while P is changed.

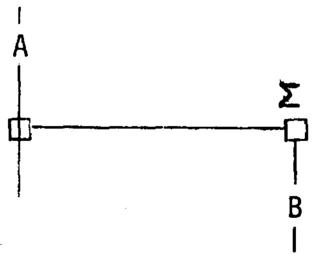


(This notation was used in A.15)

A.18: On occasion we indicate arithmetic operations on values, as in this picture. After the activity of the row occurs, C carries the value A+B.

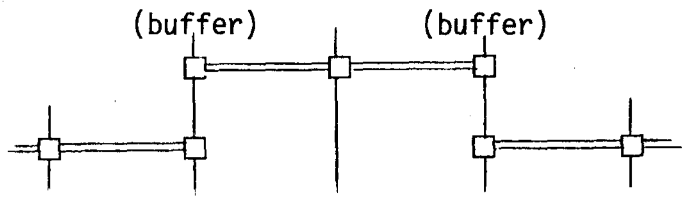


A.18.1: If A is a matrix, then B as the sum over the elements of A could be pictured as follows.



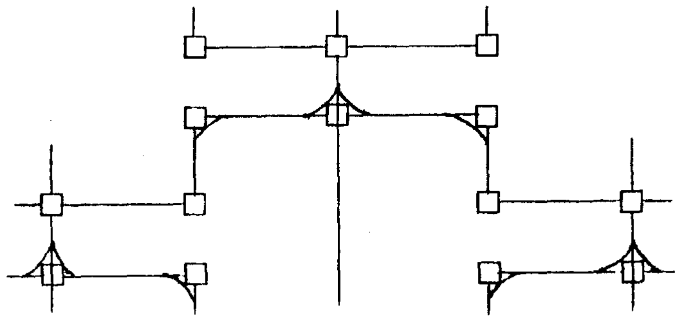
A.19: Buffered communication. A fragment of Figure 2 (of the main report is

.1.



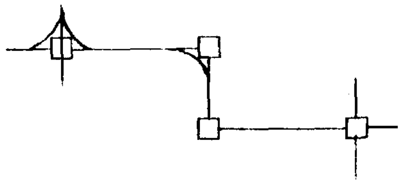
This can be expanded to

.2.



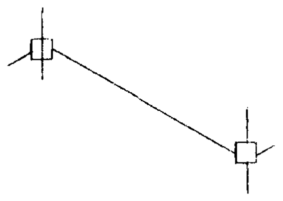
The figure .2 contains the fragment

.3



for which we introduce the abbreviated notation:

.4



The slanted bar asserts that the lower activity consumes something produced in the upper activity, and that a buffer not explicitly shown mediates the transfer from the producing to the consuming activity. With this notation, Figure 4 of the main report is transformed into Fig. 5.

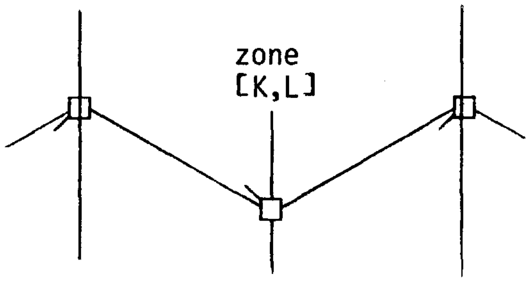
APPENDIX B

Notes on Fitting the SIMPLE Code into Role Diagrams and VAL Modules

Figure 6 of the main report somewhat schematically shows the connectivity of communication among processors, when one processor is assigned to each nodal and each zonal region of the dataflow graph. In this appendix we discuss the connections in more detail, and also discuss certain ways in which the algorithm of SIMPLE has been restated to clarify the connectivity. The objective is to help in considering hardware requirements, and to clarify aspects of the translation from FORTRAN into VAL.

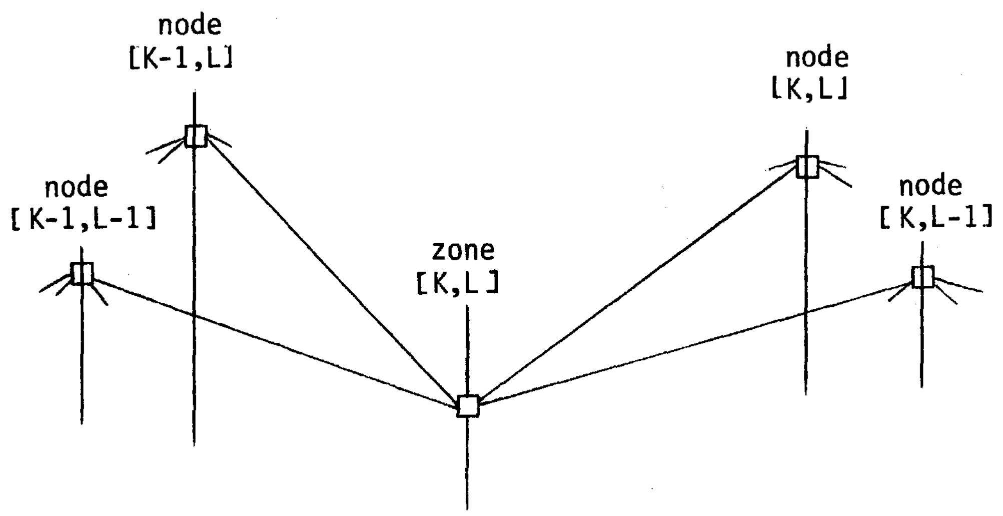
B.1. Interpretation of the cycle

Fig. 6 shows phase 3 as producing new values for zone [K,L]as follows.



.1: Schematic representation of production of zonal value.

The fragment .1 is a schematic picture of an activity at zone K,L that draws on values from the four neighboring (i.e. corner) nodes to feed into the production of new values for the zone. With the indexing convention defined in Fig. 1 of the main report, one sees that the fragment .1 stands for the connections shown in .2:



.2: Completed fragment showing all connections of nodes to a zone.

The nodal values are a vector (with R and Z components) for velocity and a vector for position at each node. The corresponding type definitions and declarations in SIMPLE_VAL are:

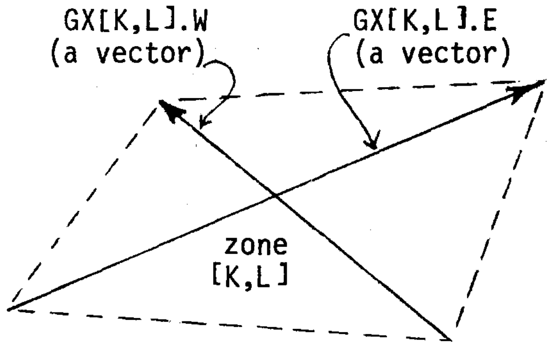
```
type vector = record[R, Z: real];
type nodal = array[array[vector]];

X, % position
V: % velocity
nodal %.
```

The correspondence between these names as used in SIMPLE_VAL and the names used in the FORTRAN code of SIMPLE is:

FORTRAN code	VAL code
R	X.R
Z	X.Z
U	V.R
W	V.Z .

In order to clarify the connectivity, as well as to eliminate some unnecessary arithmetic, we introduce auxiliary variables, starting with a kind of tensor -- GX -- that describes the diagonal dimensions of each zone:



.3: Definition of GX.

GX is, at least in spirit, a tensor; GX[K,L].W is the vector difference between the vector position at the nortwest corner and the vector position at the southeast corner. GX is produced for interior zones by ZONE_GEOM in phase 3, and for boundary zones by BOUNDARY_PROJECT; in the first case the defining relation is

```
.4.     type zone_tensor = array array record E, W: vector   ;
        GX: zone_tensor :=
           forall K in [LIM.KN+1, LIM.KX], L in [LIM.LN+1, LIM.LX] construct
           record[E: X[K,L]- X[K-1,L-1]; W: X[K-1,L]- X[K,L-1]] endall; %.
```

Note that X is a vector, so that .4 is a shorthand expression; strictly speaking one must define a subtraction function with vector arguments. This is easy to do, but clutters the presentation. With the understanding that we are abbreviating, we shall apply "-", "+" and multiplication by a scalar ("*") to vectors. The node-to-zone communications needed to form GX are shown in the picture .2. The auxiliary variable GV is a zonal tensor formed from V in exactly the same way that GX is formed from X.

Now we address phase 2 and the calculation of V. Prior to communicating from the zones around a given node to the node, a tensor STRESS is calculated for each zone; this calculation for a given zone draws only on array elements for that zone. The computation covers boundary zones, set up in phase 1, as well as interior zones.

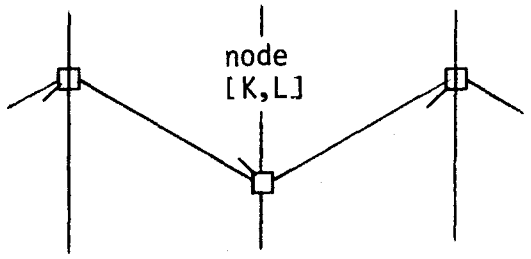
```
.5      STRESS: zone_tensor :=
            forall K in [LIM.KN, LIM.KX+1], L in[LIM.LN, LIM.LX+1] construct
            record[E: (P[K,L]+ Q[K,L])*GX[K,L].E; %scalar * vector
                   W: (P[K,L]+ Q[K,L])*GX[K,L].W ] endall ; %,
```

where P and Q are pressure and artificial viscosity, respectively, just as in the FORTRAN code. In phase 1 the auxilliary variable RHOJ is produced as:

```
.6      RHOJ : zonal :=
            forall K in[LIM.KN, LIM.KX+1], L in [LIM.LN, LIM.LX+1] construct
            RHO[K,L]*AJ[K,L] endall; %,
```

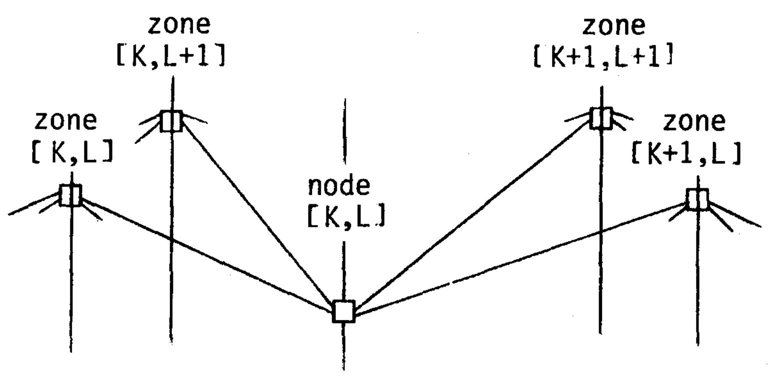
where RHO and AJ are density and area jacobian, just as in the FORTRAN code.

Phase 2 of the cycle produces new values for each node, namely V and X. The fragment that produces values for a particular node, say node K,L appears in phase 2 of Fig. 6 as follows.



.7: Schematic representation of the production of a nodal value.

The fragment .7 is a schematic picture of an activity that draws on values from the four zones around node [K,L] to feed into the production of new values of X and V for the node. Thus the fragment .7 stands for



.8: Completed fragment showing all zones connected to a node

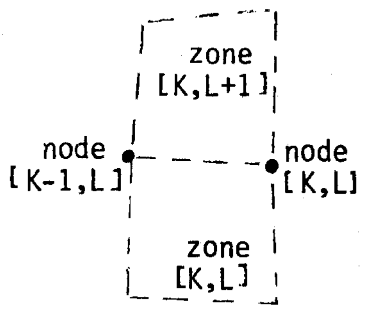
Each "cable" of values from a zone to node [K,L] must carry STRESS and RHOJ from the zone, and at least one of these cables must bring the time steps DTNPH and DTN as well. (DTNPH and DTN are used here as they are in the FORTRAN code of SIMPLE (1979).) The activity of the node in .8 during phase 2 of the cycle is to calculate an acceleration (ACC), to use this acceleration to update velocity (V), and then to use the velocity to update position (X). In updating velocity a time step DTN is used. Position times interleave the times at which velocity is calculated, so that a different time step (DTNPH) is used to update position. Continuing to use the abbreviation of scalar operation signs for operations on vector values, this activity can be expressed in VAL as:

```
V, X :=
forall K in[LIM.KN, LIM.KX+1], L in [LIM.LN, LIM.LX+1] construct
   let Y: vector := (2./(RHOJ[K,L] + RHOJ[K,L+1] + RHOJ[K+1,L] + RHOJ[K+1,L+1]))
      *(STRESS[K,L+1].E + STRESS[K,L].W - (STRESS[K+1,L+1].W + STRESS[K+1,L].E));
      ACC: vector := record[R: -Y.Z; Z: Y.R];
      V1: vector := DTN*ACC + V[K,L]
   in V1, DTNPH*V1 + X[K,L] endlet
endall
```

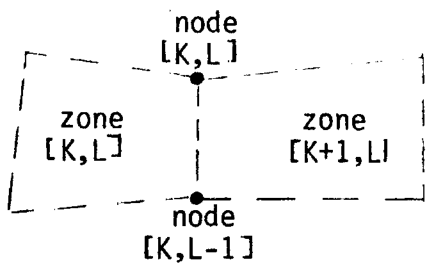
After expansion of the vector operations, this code would provide the functions VELOCITY and POSITION of Sec. 5.2.3.

Phase 4 involves arrays that are partly nodal and partly zonal in character. An element of CBB is obtained as an intermediate between two nodes of the same L-coordinate but adjoining K coordinates, and two zones bounded by the nodal K coordinates and on either side of the L coordinate:

In calculating heat conduction subroutine CONDUCT of the SIMPLE FORTRAN code generates arrays CBB and DBB, per lines 1583 through 1608. CBB and DBB draw on values from both nodes and zones, as shown:



For CBB[K,L]

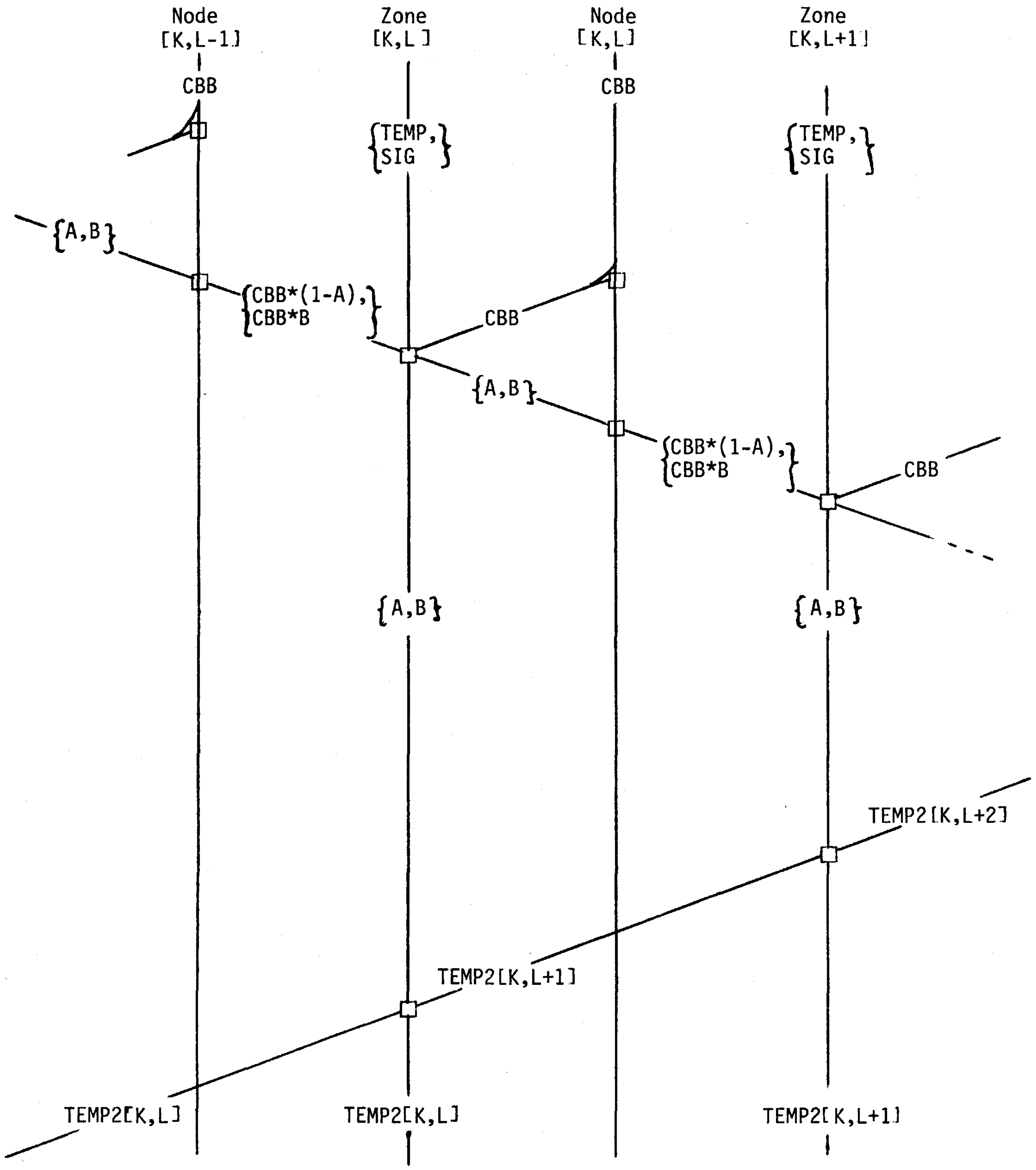


For DBB[K,L]

.9: Nodes and zones that supply values to the calculation of CBB[K,L] and DBB[K,L].

To adhere strictly to the connectivity shown in Fig. 6, one programs the calculation of CBB and DBB in two parts, one as an augmentation of ZONE_GEOM and the other as part of an augmented ENERGY_HYDRO. The augmentation consists of generating geometrical quantities as part of ZONE_GEOM, referring these to zones, as was done for GX, and then using these quantities to simplify the connectivity needed in ENERGY_HYDRO. An alternative which is displayed in SIMPLE_VAL of Sec. 5.2.3 is to accept a slightly more complex connectivity and thereby avoid the introduction of more auxilliary variables.

CBB and DBB are partly zonal and partly nodal in character, so that fitting them to either class of processors is arbitrary. Because the nodal processors are less heavily used, we have assumed that they would be used to compute CBB and DBB from zonal quantities (CC in FORTRAN). The consequent connectivity is shown in phase 4 of Fig. 6. For the Z-sweep this connectivity is shown in more detail in .10:



.10: Detail of Z-sweep of ENERGY_HEAT.

Appendix C: The SIMPLE code in FORTRAN

Edition of February 12, 1979 as provided by John Woodruff

```
$PUTT %ME,,,100000 100000,,,2000
      PROGRAM H2DD(HFILE,TAPE3=HFILE)
C
      COMMON /KLS/ K,L,DEBUG,VERSION,WHER,WHEN,P1D6,PIE,IGEN,P1D2
     X ,DTC,KC,LC,DTEN,KEN,LEN,SKE,HN,SIEL,CNN,ENC,ENH,ENCG,WN
     X   ,NCP,P1D8,VCUT
C
      COMMON /PROGG/ RO,ZO,R1,Z1,RP,ZP,RR,ZZ
C
      COMMON /COMN/   R(33,33),Z(33,33),U(33,33),RHO(33,33),Q(33,33)
     X            ,E(33,33),P(33,33),AJ(33,33),S(33,33),NBC(33,33)
     X        ,W(33,33),TEMP(33,33)
     X ,  A(33,33),B(33,33),CC(33,33),DUM(33,33),CBB(33,33)
     X ,  DBB(33,33),CAP(33,33),SIG(33,33),TS(33,33)
C
      COMMON /PARAM/ NYCL,TNUP,DTNUP,DTN,DTNPH,DTNMH,EDTIME,EDDT
     X       ,GAM,GAMZ,COF,C1F,C1,TMAX,DTMAX,DTMIN,TFLR,NOHYD
     X       ,C2,P2,P3,NO,NTTY,NED
C
      COMMON /KLSPACE/ KMN,LMN,KMX,LMX,KMXZ,LMXZ,KMNP,LMNP,KMXP,LMXP
C
      COMMON /GENCOM/ RHOO,EO,UO,PO,WO,DR,DZ,NBCU,NBCD,NBCL,NBCR
     X ,PB(3),PBB(3),QB(3)
C
      COMMON /MINMAX/ XMIN,XMAX,YMIN,YMAX,PMIN,PMAX,QMIN,QMAX
     X,RMIN,RMAX,KQ,LQ,KR,LR,KP,LP
     X ,XMINX,XMAXX,YMINX,YMAXX
C
      COMMON /TIMING/ NBT(20),NCT(20),NET(20),NPT(20),NXT(20)

      COMMON /EOSCOM/ KEOS,TARG1,TARG2,TARG3,RARG1,RARG2,RARG3,
     X  FUNC1,FUNC2,FUNC3,TEMPS,EPS,EPSO

      COMMON /COM2/ NTSV(2),NRSV(2),MSV(2),TES(7),RES(9)
     X ,AES(12),BES(12),CES(12),DES(12),EES(12),FES(12),GES(12)
     X ,HES(12),PES(12),ITES(3),IRES(3),IZES(3)
C
C NCYL = CYCLE COUNTER EDTIME= TIME OT EDIT
C TNUP = PROBLEM TIME(N+1) EDDT = DELTAT NEXT EDIT
C DTN = DELTAT (N) TMAX = MAXIMUM TIME
C DTNPH= DELTAT (N+1/2) DTMAX = MAXIMUM ALLOWED DT
C DTNMH= DELTAT (N-1/2) DTMIN = MINIMUM ALLOWED DT
      DIMENSION ARRAY(1)
      EQUIVALENCE (ARRAY,R)
      DATA I1/0/
      DATA NLINKS/5/
      DATA VERSION /1./
      DATA NCP/10/
      DATA IER/0/
      DATA NTTY/59/
      DATA NO/3/
      DATA DEBUG/0./
      DATA DTMAX/.01/
      DATA DTMIN /.0001/
      DATA TFLR/.0001/
      DATA NOHYD/0/
      DATA PIE/3.1415926535898/
      DATA EDTIME/0./
      DATA EDDT/4./
      DATA P1D2/.5/
      DATA TMAX/12.001/
      DATA VCUT/1.E-10/
      DATA DTEN/1.E+10/
      DATA DTC/1.E+10/
C
      CALL CHANGE(2H+H)
      CALL ASSIGN(3,2RPH)
      CALL CLOCK(WHER,WHEN)
C
C  ZERO OUT ALL ARRAYS
C
      L=21*33*33
C
      DO 10 K=1,L
C
      ARRAY(K)=0.
C
   10 CONTINUE
C
C  SET UP EOS TABLES
C
      CALL SETUP
C
C  SET PARAMETERS FOR TEST PROBLEM
C
      GAM=1.4
      P2=6.
      P3=0.
      PB(1)=1.
      PB(2)=0.
      PB(3)=0.
      QB(1)=1.
      QB(2)=0.
      QB(3)=0.
      PBB(1)=0.
      PBB(2)=P2
      PBB(3)=P3
      RHOO=1.4
      KMN=2
      LMN=2
      KMX=5
      LMX=22
      DR=1.
      DZ=1.
      DTNPH=.01
      TMAX=10.
      DTN=DTNPH
      DTNMH=DTNPH
      TNUP=0.
      E0=0.
      U0=0.
      P0=0.
      W0=0.
      NBCU=1
      NBCR=2
      NBCL=1
      NBCD=1
      C2=1.5
      COF=C2*.25
      C1=.5
      C1F=.5*C1
      GAMZ=GAM-1.
      DTC=100.
      P1D8=1./8.
      HN=0.
      WN=0.
C
C  GET INPUT PARAMETERS
C
      WRITE(NTTY,4)
    4 FORMAT(23H ENTER INPUT PARAMETERS)
C
      READ(NTTY,5)KMN,KMX,LMN,LMX,EDDT,EDTIME,TMAX
    5 FORMAT(4I2,3F5.2)
C
      KMNP=KMN+1
      LMNP=LMN+1
C
      KMXP=KMX+1
      LMXP=LMX+1
C
      KMXZ=KMX-1
      LMXZ=LMX-1
C
C  GENERATE PROBLEM
C
      CALL GEN
C
      IGEN=0
C
C  INITIALIZE TIMER
C
      NED=1
      I2=NECOND(I1)
C
C START CYCLE HERE
C
  1   CONTINUE
C
      DTC2=1.E+12
      SKE=0.
      ENC=0.
      ENH=0.
      DTEN=1.E+12
C
C*******************************************
C*                                         *
C*  GEOMETRY CALCULATION FOR BOUNDARY ZONES  *
C*                                         *
C*******************************************
C
C  SET UP BOTTOM SIDE BOUNDARY ZONES
C
C  P(K,L+1)
C  O(K,L)     1(K+1,L)
C  R(K,L-1)
C
      L=LMN
C
      RO=R(KMN,L)
      ZO=Z(KMN,L)
C
      DO 200 K=KMN,KMXZ
C
      R1=R(K+1,L)
      Z1=Z(K+1,L)
C
      RP=R(K,L+1)
      ZP=Z(K,L+1)
C
      CALL PROJCT
C
      R(K,L-1)=RR
      Z(K,L-1)=ZZ
C
      RO=R1
      ZO=Z1
C
  200 CONTINUE
C
C  SET UP BOTTOM RIGHT CORNER
C
C            P(K,L+1)
C  1(K-1,L)  O(K,L)
C            R(K,L-1)
C
      K=KMX
      L=LMN
C
      RO=R(K,L)
      ZO=Z(K,L)
C
      R1=R(K-1,L)
      Z1=Z(K-1,L)
C
      RP=R(K,L+1)
      ZP=Z(K,L+1)
C
      CALL PROJCT
C
      R(K,L-1)=RR
      Z(K,L-1)=ZZ
C
C  SET UP TOP SIDE BOUNDARY ZONES
C
C  R(K,L+1)
C  O(K,L)    1(K+1,L)
C  P(K,L-1)
C
      L=LMX
      RO=R(KMN,L)
      ZO=Z(KMN,L)
C
      DO 204 K=KMN,KMXZ
C
      R1=R(K+1,L)
      Z1=Z(K+1,L)
C
      RP=R(K,L-1)
      ZP=Z(K,L-1)
```

```
C
      CALL PROJCT
C
      R(K,L+1)=RR
      Z(K,L+1)=ZZ
C
      RO=R1
      ZO=Z1
C
  204 CONTINUE
C
C  SET UP TOP RIGHT CORNER
C
C            R(K,L+1)
C  1(K-1,L)  O(K,L)
C            P(K,L-1)
C
      K=KMX
      L=LMX
C
      RO=R(K,L)
      ZO=Z(K,L)
C
      R1=R(K-1,L)
      Z1=Z(K-1,L)
C
      RP=R(K,L-1)
      ZP=Z(K,L-1)
C
      CALL PROJCT
C
      R(K,L+1)=RR
      Z(K,L+1)=ZZ
C
C  SET UP LEFT SIDE BOUNDARY ZONES
C
C            1(K,L+1)
C  R(K-1,L)  O(K,L)    P(K+1,L)
C
      K=KMN
      RO=R(K,LMN)
      ZO=Z(K,LMN)
C
      DO 207 L=LMN,LMXZ
C
      R1=R(K,L+1)
      Z1=Z(K,L+1)
C
      RP=R(K+1,L)
      ZP=Z(K+1,L)
C
      CALL PROJCT
C
      R(K-1,L)=RP
      Z(K-1,L)=ZZ
C
      RO=R1
      ZO=Z1
C
  207 CONTINUE
```

```
C
C  SET UP TOP LEFT CORNER
C
C  R(K-1,L)  O(K,L)    P(K+1,L)
C            1(K,L-1)
C
      K=KMN
      L=LMX
C
      RO=R(K,L)
      ZO=Z(K,L)
C
      R1=R(K,L-1)
      Z1=Z(K,L-1)
C
      RP=R(K+1,L)
      ZP=Z(K+1,L)
C
      CALL PROJCT
C
      R(K-1,L)=RR
      Z(K-1,L)=ZZ
C
C  SET UP RIGHT SIDE BOUNDARY ZONES
C
C            1(K,L+1)
C  P(K-1,L)  O(K,L)    R(K+1,L)
C
      K=KMX
      RO=R(K,LMN)
      ZO=Z(K,LMN)
C
      DO 210 L=LMN,LMXZ
C
      R1=R(K,L+1)
      Z1=Z(K,L+1)
C
      RP=R(K-1,L)
      ZP=Z(K-1,L)
C
      CALL PROJCT
C
      R(K+1,L)=RR
      Z(K+1,L)=ZZ
C
      RO=R1
      ZO=Z1
C
  210 CONTINUE
C
C  SET UP TOP RIGHT CORNER
C
C  P(K-1,L)  O(K,L)    R(K+1,L)
C            1(K,L-1)
C
      K=KMX
      L=LMX
C
      RO=R(K,L)
      ZO=Z(K,L)
```

```
C
      R1=R(K,L-1)
      Z1=Z(K,L-1)
C
      RP=R(K-1,L)
      ZP=Z(K-1,L)
C
      CALL PROJCT
C
      R(K+1,L)=RR
      Z(K+1,L)=ZZ
C
C  SET UP TOP RIGHT CORNER
C
C  P(K-1,L+1)  1(K,L+1)  R(K+1,L+1)
C              O(K,L)
C
      K=KMX
      L=LMX
C
      RO=R(K,L)
      ZO=Z(K,L)
C
      R1=R(K,L+1)
      Z1=Z(K,L+1)
C
      RP=R(K-1,L+1)
      ZP=Z(K-1,L+1)
C
      CALL PROJCT
C
      R(K+1,L+1)=RR
      Z(K+1,L+1)=ZZ
C
C  SET UP BOTTOM LEFT CORNER
C
C              O(K,L)
C  R(K-1,L-1)  1(K,L-1)  P(K+1,L-1)
C
      K=KMN
      L=LMN
C
      RO=R(K,L)
      ZO=Z(K,L)
C
      R1=R(K,L-1)
      Z1=Z(K,L-1)
C
      RP=R(K+1,L-1)
      ZP=Z(K+1,L-1)
C
      CALL PROJCT
C
      R(K-1,L-1)=RR
      Z(K-1,L-1)=ZZ
C
C  SET UP BOTTOM RIGHT CORNER
C
C          P(K+1,L+1)
C  O(K,L)  1(K+1,L)
```

```
C          R(K+1,L-1)
C
      K=KMX
      L=LMN
C
      RO=R(K,L)
      ZO=Z(K,L)
C
      R1=R(K+1,L)
      Z1=Z(K+1,L)
C
      RP=R(K+1,L+1)
      ZP=Z(K+1,L+1)
C
      CALL PROJCT
C
      R(K+1,L-1)=RR
      Z(K+1,L-1)=ZZ
C
C  SET UP TOP LEFT CORNER
C
C  R(K-1,L+1)
C  1(K-1,L)     O(K,L)
C  P(K-1,L-1)
C
      L=LMX
      K=KMN
C
      RO=R(K,L)
      ZO=Z(K,L)
C
      R1=R(K-1,L)
      Z1=Z(K-1,L)
C
      RP=R(K-1,L-1)
      ZP=Z(K-1,L-1)
C
      CALL PROJCT
C
      R(K-1,L+1)=RR
      Z(K-1,L+1)=ZZ
C
C************************************
C*                                  *
C*   SET UP BOUNDARY ZONE ATTRIBUTES  *
C*                                  *
C************************************
C
C  SET UP BOTTOM SIDE BOUNDARY ZONES
C
C  (K,L) = (K,L+1)
C
      L=LMN
C
      DO 255 K=KMNP,KMX
C
      RHO(K,L)=RHO(K,L+1)
      AJ(K,L)=AJ(K,L+1)
      IP=NBC(K-1,L)
      Q(K,L)=QB(IP)*Q(K,L+1)
      P(K,L)=PBB(IP)+PB(IP)*P(K,L+1)
C
  255 CONTINUE
C
C  SET UP RIGHT SIDE BOUNDARY ZONES
C
C  (K+1,L) = (K,L)
C
      K=KMX
C
      DO 265 L=LMNP,LMX
C
      RHO(K+1,L)=RHO(K,L)
      AJ(K+1,L)=AJ(K,L)
      IP=NBC(K,L)
      Q(K+1,L)=QB(IP)*Q(K,L)
      P(K+1,L)=PBB(IP)+PB(IP)*P(K,L)
C
  265 CONTINUE
C
C  SET UP TOP SIDE BOUNDARY ZONES
C
C  (K,L+1) = (K,L)
C
      L=LMX
C
      DO 275 K=KMNP,KMX
C
      RHO(K,L+1)=RHO(K,L)
      AJ(K,L+1)=AJ(K,L)
      IP=NBC(K-1,L)
      Q(K,L+1)=QB(IP)*Q(K,L)
      P(K,L+1)=PBB(IP)+PB(IP)*P(K,L)
C
  275 CONTINUE
C
C  SET UP LEFT SIDE BOUNDARY ZONES
C
C  (K,L) = (K+1,L)
C
      K=KMN
C
      DO 285 L=LMNP,LMX
C
      RHO(K,L)=RHO(K+1,L)
      AJ(K,L)=AJ(K+1,L)
      IP=NBC(K,L-1)
      Q(K,L)=QB(IP)*Q(K+1,L)
      P(K,L)=PBB(IP)+PB(IP)*P(K+1,L)
C
  285 CONTINUE
C
C  SET UP BOTTOM LEFT CORNER
C
      P(KMN,LMN)=P(KMNP,LMNP)
      Q(KMN,LMN)=Q(KMNP,LMNP)
      RHO(KMN,LMN)=RHO(KMNP,LMNP)
      AJ(KMN,LMN)=AJ(KMNP,LMNP)
C
C  SET UP BOTTOM RIGHT CORNER
```

```
C
      P(KMXP,LMN)=P(KMXP,LMN+1)
      Q(KMXP,LMN)=Q(KMXP,LMN+1)
      RHO(KMXP,LMN)=RHO(KMXP,LMN+1)
      AJ(KMXP,LMN)=AJ(KMXP,LMN+1)
C
C  SET UP TOP RIGHT CORNER
C
      P(KMXP,LMXP)=P(KMXP,LMX)
      Q(KMXP,LMXP)=Q(KMXP,LMX)
      RHO(KMXP,LMXP)=RHO(KMXP,LMX)
      AJ(KMXP,LMXP)=AJ(KMXP,LMX)
C
C  SET UP TOP LEFT CORNER
C
      P(KMN,LMXP)=P(KMNP,LMXP)
      Q(KMN,LMXP)=Q(KMNP,LMXP)
      RHO(KMN,LMXP)=RHO(KMNP,LMXP)
      AJ(KMN,LMXP)=AJ(KMNP,LMXP)
C
C  GET BOUNDARY CONDITION COMPUTE TIME
C
      I2=NECOND(I1)
      NBT(NED)=NBT(NED)+I2
C
C  DEBUG EDIT
C
      IF(DEBUG.EQ.0.) GO TO 442
      IGEN=1
C
      WRITE(NO,441)
  441 FORMAT(9H DEBUG  1)
C
      CALL EDIT
C
  442 CONTINUE
C
      DO 450 L=LMN,LMX
      DO 445 K=KMN,KMX
COMPUTE ACCELERATION
      AU=(P(K,L)+Q(K,L)) * (Z(K,L-1)-Z(K-1,L)) +
     X    (P(K+1,L)+Q(K+1,L))*(Z(K+1,L)-Z(K,L-1)) +
     X    (P(K+1,L+1)+Q(K+1,L+1))*(Z(K,L+1)-Z(K+1,L)) +
     X    (P(K,L+1)+Q(K,L+1))*(Z(K-1,L)-Z(K,L+1))
      AW=(P(K,L)+Q(K,L)) * (R(K,L-1)-R(K-1,L)) +
     X    (P(K+1,L)+Q(K+1,L)) * (R(K+1,L)-R(K,L-1)) +
     X    (P(K+1,L+1)+Q(K+1,L+1)) * (R(K,L+1)-R(K+1,L)) +
     X    (P(K,L+1)+Q(K,L+1)) * (R(K-1,L)-R(K,L+1))
      AUW=RHO(K,L)*AJ(K,L)+RHO(K+1,L)*AJ(K+1,L)+RHO(K,L+1)*AJ(K,L+1)
     X    +RHO(K+1,L+1)*AJ(K+1,L+1)
      AUW=2./AUW
      AU=-AU*AUW
      AW=AW*AUW
      U(K,L)=U(K,L)+DTN*AU
C ADVANCE VELOCITIES TO N+1/2 FROM N-1/2
      W(K,L)=W(K,L)+DTN*AW
C POSITION (N+1)
      IF(ABS(U(K,L)).LE.VCUT)U(K,L)=0.
      IF(ABS(W(K,L)).LE.VCUT)W(K,L)=0.
      A(K,L)=U(K,L)**2+W(K,L)**2
  445 CONTINUE
  450 CONTINUE
      IF(NOHYD.EQ.1) GO TO 455
C NOHYD=1 TO SKIP HYDRO
      DO 452 L=LMN,LMX
      DO 451 K=KMN,KMX
      R(K,L)=R(K,L)+DTNPH*U(K,L)
      Z(K,L)=Z(K,L)+DTNPH*W(K,L)
  451 CONTINUE
  452 CONTINUE
C     ACCELERATION VELOCITY AND
C     CO-ORDINATES DONE            END OF NECOND PASS
C
C                      BEGIN LOOP 3
C TEMPRY *********
C
C  DEBUG EDIT
C
      IF(DEBUG.EQ.0.) GO TO 455
      IGEN=1
C
      WRITE(NO,456)
  456 FORMAT(9H DEBUG  2)
C
      CALL EDIT
C
  455 CONTINUE
C
      CALL HWORK
C
C COMPUTE HYDRO WORK ON THE BOUNDARY
      DO 490 L=LMNP,LMX
      DO 485 K=KMNP,KMX
      AJ1=R(K,L)*  (Z(K-1,L)-Z(K,L-1)) +
     X    R(K-1,L)*  (Z(K,L-1)-Z(K,L)) +
     X    R(K,L-1)*(Z(K,L)-Z(K-1,L))
      AJ3=R(K-1,L)*  (Z(K-1,L-1)-Z(K,L-1)) +
     X    R(K-1,L-1)*(Z(K,L-1)-Z(K-1,L)) +
     X    R(K,L-1)*(Z(K-1,L)-Z(K-1,L-1))
C
C     JACOBIAN AREA IN (R,Z) PLANE
C
      SN=S(K,L)
      AJ(K,L)=P1D2*(AJ1+AJ3)
      S(K,L)=P1D6*((R(K,L)+R(K-1,L)+R(K,L-1))*AJ1 +
     X             (R(K-1,L)+R(K-1,L-1)+R(K,L-1))*AJ3 )
C
C     S=VOLUME/2X (CM**3/RADIAN)
      VN=1./RHO(K,L)
C VN=SPECIFIC VOLUME AT (N)
C     VNP=SPECIFIC VOLUME AT (N+1)
      RHO(K,L)=RHO(K,L)*SN/S(K,L)
       DUM(K,L)=RHO(K,L)*S(K,L)
C DUM=MASS
C     DENSITY AT N+1
      VNP=1./RHO(K,L)
      DELV=VNP-VN
COMPUTE ARTIFICIAL VISCOSITY
      DRK=R(K,L)-R(K-1,L-1)+R(K,L-1)-R(K-1,L)
      DRL=R(K,L)-R(K-1,L-1)+R(K-1,L)-R(K,L-1)
      DZK=Z(K,L)-Z(K-1,L-1)+Z(K,L-1)-Z(K-1,L)
      DZL=Z(K,L)-Z(K-1,L-1)+Z(K-1,L)-Z(K,L-1)
      DUK=U(K,L)-U(K-1,L-1)+U(K,L-1)-U(K-1,L)
      DUL=U(K,L)-U(K-1,L-1)+U(K-1,L)-U(K,L-1)
      DWK=W(K,L)-W(K-1,L-1)+W(K,L-1)-W(K-1,L)
      DWL=W(K,L)-W(K-1,L-1)+W(K-1,L)-W(K,L-1)
C
C     DRK=2DR/DK
C     DRL=2DR/DL
C
      W1= DRK*DWL-DZK*DUL
      W2= DUK*DZL-DWK*DRL
      Q(K,L)=0.
      W3=0.
      W4=0.
      IF(W1.LT.0.)W3=W1**2/(DRK**2+DZK**2)
      IF(W2.LT.0.)W4=W2**2/(DRL**2+DZL**2)
      IF((W3+W4).EQ.0.) GO TO 465
      CA=SQRT(GAM*P(K,L)/RHO(K,L))
C DON'T COMPUTE Q IF ZONE IS NOT BEING COMPRESSED
      Q(K,L)=COF*RHO(K,L)*(W3+W4) + C1F*CA*RHO(K,L)*SQRT(W3+W4)
C  C1F= C1*.5    COF=.25*CO**2    CA=SOUND SPEED
C
      IF(CA.EQ.0.) GO TO 465
      TSO=(AJ(K,L)**2)/(CA*CA*(DRK**2+DRL**2+DZK**2+DZL**2))
      IF(DTC2.LE.TSO) GO TO 462
C HAVE A NEW MINIMUM DELTA T
      DTC2=TSO
      KC=K
      LC=L
  462 CONTINUE
  465 CONTINUE
      EPS=E(K,L)-(P(K,L)+Q(K,L))*DELV
C  E(N+1)
C
      RARG1=RHO(K,L)
      CALL TEMPCAL
      TARG1=TEMPS
      CALL IES1
      PNP=FUNC1
C GAMMA-LAW EOS   GAMZ=GAM-1.
      E(K,L)=E(K,L)-(.5*(PNP+P(K,L))+Q(K,L))*DELV
      E(K,L)=AMAX1(E(K,L),1.E-30)
      EPS=E(K,L)
      CALL TEMPCAL
C GET TEMPERATURE  AS FUNCTION OF E.RHO
      TARG1=AMAX1(TEMPS,TFLR)
       TEMP(K,L)=TARG1
      CALL IES1
C GET PRESSURE
      P(K,L)=FUNC1
C  E(N+1)
C  P(N+1)
C
      SKE=SKE+P1D8*DUM(K,L)*(A(K,L)+A(K-1,L)+A(K,L-1)+A(K-1,L-1))
C KINETIC ENERGY FOR THE ZONE
  485 CONTINUE
  490 CONTINUE
C********************************END OF LOOP 3
C
```

```
C  DEBUG EDIT
C
      IF(DEBUG.EQ.0.) GO TO 495
      IGEN=1
C
      WRITE(NO,493)
  493 FORMAT(9H DEBUG  3)
C
      CALL EDIT
C
  495 CONTINUE
C
      IGEN=0
      I2=NECOND(I1)
      NXT(NED)=NXT(NED)+I2
C
      CALL CONDUCT
C                       DO HEAT CONDUCTION
C
      NYCL=NYCL+1
C ADVANCE CYCLE COUNTER
      DTNMH=DTNPH
      DTC=SQRT(DTC2)
      DTNPH=DTC
      DTNPH=AMIN1(DTNPH,DTEN,DTMAX)
C LIMIT MAGNITUDE OF DT
      DTN=.5*(DTNPH+DTNMH)
      TNUP=TNUP+DTNPH
      IF(DTNPH.GE.DTMIN) GO TO 602
C********** DT IS BELOW ALLOWED MINIMUM **********
      WRITE(NO,601)NYCL,TNUP,DTNPH,DTMIN
      WRITE(NO,601)NYCL,TNUP,DTNPH,DTMIN
  601 FORMAT(12H DTSTOP NYCL,I6,3H T ,E12.4,4H DT ,E12.4)
      GO TO 999
  602 CONTINUE
      TE=SKE+ENH
      CN=TE-HN-WN
      IF(NYCL.EQ.1) CNOLD=CN
      CNN=CN-CNOLD
      ENCG=ENCG+CNN
      CNOLD=CN
      IF(MOD(NYCL,NCP).NE.0) GO TO 603
      WRITE(NO,706)
  706 FORMAT(6H CYCLE,4X,5HTIME ,7X,2HDT,10X,3HDTC,5X,8H  KC  LC,
     X        4X,3HDTE,5X,8H KEN LEN)
      WRITE(NO,707)NYCL,TNUP,DTNPH,DTC,KC,LC,DTEN,KEN,LEN
  707 FORMAT(I6,3E12.4,2I4,E12.4,2I4)
      Z2=ABS(ENC-(ENH+HN))/ENC
      WRITE(NO,708)
  708 FORMAT(4X,4HETOT,8X,4HIE  ,8X,4HKE  ,8X,4HHN  ,8X,4HWN  ,
     X        8X,5HECONS,7X,5HCN(N),7X,4HECNG)
      WRITE(NO,709)TE,ENC,SKE,HN,WN,Z2,CNN,ENCG
  709 FORMAT(8E12.4)
 603  CONTINUE
      I2=NECOND(I1)
      NCT(NED)=NCT(NED)+I2
C RUN TIME FOR PHYSICS
      IF(TNUP.LT.EDTIME) GO TO 605
C TIME TO EDIT
      CALL EDIT
      WRITE(NO,604)NYCL,TNUP,DTNPH,RMAX,KR,LR
  604 FORMAT(12H EDIT NYCL= ,I6,2E12.4,E14.5,2I4)
C MESSAGE TO TTY
      EDTIME=EDTIME+EDDT
C ADVANCE EDITME TO NEXT VALUE
  605 CONTINUE
      IF(TNUP.LT.TMAX) GO TO 610
C********** PROBLEM HAS REACHED TMAX**********
      WRITE(NO,607)NYCL,TNUP,TMAX
      WRITE(NO,607)NYCL,TNUP,TMAX
  607 FORMAT(12H STOP TMAX  ,I6,2E12.4)
      GO TO 999
  610 CONTINUE
      GO TO 1
  999 CONTINUE
C PROBLEM COMPLETED GET OFF
C     CALL PLOTE
      WRITE(NO,616)(NBT(K),NCT(K),NET(K),NPT(K),NXT(K),K=1,NED)
  616 FORMAT(5(1X,I10))
      I1=0
      DO 618 K=1,NED
      I1=I1+NBT(K)+NCT(K)+NET(K)+NPT(K)+NXT(K)
  618 CONTINUE
C
C
      IF(I1.EQ.0) I1=1
      DO 619 K=2,NED
      NBT(1)=NBT(1)+NBT(K)
      NCT(1)=NCT(1)+NCT(K)
      NET(1)=NET(1)+NET(K)
      NPT(1)=NPT(1)+NPT(K)
      NXT(1)=NXT(1)+NXT(K)
 619  CONTINUE
C
      AES(1)=(NBT(1)*100)/I1
      AES(2)=(NCT(1)*100)/I1
      AES(3)=(NET(1)*100)/I1
      AES(4)=(NPT(1)*100)/I1
      AES(5)=(NXT(1)*100)/I1
      WRITE(NO,620) I1,(AES(I2),I2=1,5)
  620 FORMAT(I10,5E12.4)
C
      RETURN
      END


      SUBROUTINE GEN
C
C  THIS SUBROUTINE GENERATES THE INITIAL PROBLEM TO BE RUN
C
      COMMON /KLS/ K,L,DEBUG,VERSION,WHER,WHEN,P1D6,PIE,IGEN,P1D2
     X ,DTC,KC,LC,DTEN,KEN,LEN,SKE,HN,SIEL,CNN,ENC,ENH,ENCG,WN
     X  ,NCP
C
      COMMON /PROGG/ RO,ZO,R1,Z1,RP,ZP,RR,ZZ
C
      COMMON /COMN/  R(33,33),Z(33,33),U(33,33),RHO(33,33),Q(33,33)
     X           ,E(33,33),P(33,33),AJ(33,33),S(33,33),NBC(33,33)
     X       ,W(33,33),TEMP(33,33)
     X , A(33,33),B(33,33),CC(33,33),DUM(33,33),CBB(33,33)
     X ,  DBB(33,33),CAP(33,33),SIG(33,33),TS(33,33)
C
      COMMON /PARAM/ NYCL,TNUP,DTNUP,DTN,DTNPH,DTNMH,EDTIME,EDDT
     X       ,GAM,GAMZ,COF,C1F,C1,TMAX,DTMAX,DTMIN,TFLR,NOHYD
     X       ,C2,P2,P3,NO,NTTY,NED
C
      COMMON /KLSPACE/ KMN,LMN,KMX,LMX,KMXZ,LMXZ,KMNP,LMNP,KMXP,LMXP
C
      COMMON /GENCOM/ RHOO,EO,UO,PO,WO,DR,DZ,NBCU,NBCD,NBCL,NBCR
     X ,PB(3),PBB(3),QB(3)
C
      COMMON /MINMAX/ XMIN,XMAX,YMIN,YMAX,PMIN,PMAX,QMIN,QMAX
     X,RMIN,RMAX,KQ,LQ,KR,LR,KP,LP
     X ,XMINX,XMAXX,YMINX,YMAXX
C
C  IGEN NOT EQUAL 0 WILL CAUSE THE EDIT ROUTINE TO PRINT ALL THE VARIABL
C
      DATA IGEN/1/
C
C**********************
C*                   *
C*  GENERATE NBC ARRAY  *
C*                   *
C**********************
C
C  SET BOTTOM AND TOP BOUNDARY CONDITIONS
C
      DO 52 K=KMN,KMX
C
      NBC(K,LMN)=NBCD
      NBC(K,LMX)=NBCU
C
   52 CONTINUE
C
C  SET LEFT AND RIGHT BOUNDARY CONDITIONS
C
      DO 54 L=LMN,LMX
C
      NBC(KMN,L)=NBCL
      NBC(KMX,L)=NBCR
C
   54 CONTINUE
C
C******************************************
```

```
C*                                      *
C*  GENERATE COORDINATES AND VELOCITIES *
C*                                      *
C****************************************
C
C  INITIALIZE THE MINIMUM AND MAXIMUM VALUES OF R AND Z
C
      XMINX=1.E+6
      XMAXX=-1.E+6
C
      YMINX=1.E+6
      YMAXX=-1.E+6
C
      RP=LMX-LMN
      ZP=KMX-KMN
C
      DO 58 K=KMN,KMX
C
      Z1=10+K-KMN
C
      DO 57 L=LMN,LMX
C
C  COMPUTE THE COORDINATES R AND Z
C
      RR=L-2
      ZZ=(-.5+RR/RP)*PIE
C
      R(K,L)=Z1*COS(ZZ)
      Z(K,L)=Z1*SIN(ZZ)
C
C  FIND THE MINIMUM AND MAXIMUM VALUES OF R AND Z
C
      XMINX=AMIN1(XMINX,R(K,L))
      XMAXX=AMAX1(XMAXX,R(K,L))
C
      YMINX=AMIN1(YMINX,Z(K,L))
      YMAXX=AMAX1(YMAXX,Z(K,L))
C
   57 CONTINUE
C
   58 CONTINUE
C
C**********************************************************
C*                                                        *
C*  GENERATE ZONE QUANTITIES RHO, P, E AND COMPUTE AREA   *
C*                                                        *
C**********************************************************
C
      P1D6=1./6.
C
      DO 65 L=LMNP,LMX
C
      DO 63 K=KMNP,KMX
C
      RHO(K,L)=RHOO
      P(K,L)=PO
      E(K,L)=EO
C
C  COMPUTE JACOBIAN
C
```

```
      AJ1=R(K,L)*(Z(K-1,L)-Z(K,L-1))+R(K-1,L)*(Z(K,L-1)-Z(K,L))
     X    +R(K,L-1)*(Z(K,L)-Z(K-1,L))
C
      AJ3=R(K-1,L)*(Z(K-1,L-1)-Z(K,L-1))+R(K-1,L-1)*(Z(K,L-1)-Z(K-1,L))
     X    +R(K,L-1)*(Z(K-1,L)-Z(K-1,L-1))
C
      AJ(K,L)=P1D2*(AJ1+AJ3)
C
      S(K,L)=P1D6*((R(K,L)+R(K-1,L)+R(K,L-1))*AJ1 +
     X             (R(K,L-1)+R(K-1,L)+R(K-1,L-1))*AJ3)
C
  63  CONTINUE
C
  65  CONTINUE
C
C****************
C*             *
C*  DEBUG EDIT *
C*             *
C****************
C
      IF(DEBUG.EQ.0.)GO TO 80
C
C  PRINT NBC BOUNDARY SENTINELS
C
      WRITE(NO,71)(NBC(K,LMN),K=KMN,KMX)
   71 FORMAT(3HLMN,80I1)
C
      WRITE(NO,72)(NBC(K,LMX),K=KMN,KMX)
   72 FORMAT(3HLMX,80I1)
C
      WRITE(NO,73)(NBC(KMN,L),L=LMN,LMX)
   73 FORMAT(3HKMN,80I1)
C
      WRITE(NO,74)(NBC(KMX,L),L=LMN,LMX)
   74 FORMAT(3HKMX,80I1)
C
      CALL EDIT
C
   80 CONTINUE
C
      WRITE(NO,85)
      WRITE(NTTY,85)
   85 FORMAT(21H GENERATION COMPLETED)
C
      RETURN
      END
      SUBROUTINE EDIT
C
      COMMON /KLS/ K,L,DEBUG,VERSION,WHER,WHEN,P1D6,PIE,IGEN,P1D2
     X ,DTC,KC,LC,DTEN,KEN,LEN,SKE,HN,SIEL,CNN,ENC,ENH,ENCG,WN
     X  ,NCP
C
      COMMON /COMN/  R(33,33),Z(33,33),U(33,33),RHO(33,33),Q(33,33)
     X           ,E(33,33),P(33,33),AJ(33,33),S(33,33),NBC(33,33)
     X       ,W(33,33),TEMP(33,33)
     X , A(33,33),B(33,33),CC(33,33),DUM(33,33),CBB(33,33)
     X , DBB(33,33),CAP(33,33),SIG(33,33),TS(33,33)
C
      COMMON /PARAM/ NYCL,TNUP,DTNUP,DTN,DTNPH,DTNMH,EDTIME,EDDT
     X      ,GAM,GAMZ,COF,C1F,C1,TMAX,DTMAX,DTMIN,TFLR,NOHYD
     X      ,C2,P2,P3,NO,NTTY,NED
C
      COMMON /KLSPACE/ KMN,LMN,KMX,LMX,KMXZ,LMXZ,KMNP,LMNP,KMXP,LMXP
C
      COMMON /MINMAX/ XMIN,XMAX,YMIN,YMAX,PMIN,PMAX,QMIN,QMAX
     X,RMIN,RMAX,KQ,LQ,KR,LR,KP,LP
     X ,XMINX,XMAXX,YMINX,YMAXX
C
      COMMON /TIMING/ NBT(20),NCT(20),NET(20),NPT(20),NXT(20)
C
C
C
C TEMPIS SUBROUTINE EDITS ALL MESH VARIABLES
      DATA N100/100/
      I1=0
C
C  INITIALIZE MINIMUM AND MAXIMUM VALUES OF RHO, P, Q, R AND Z
C
      RMIN=1.E+6
      RMAX=-1.E+6
C
      PMIN=1.E+6
      PMAX=-1.E+6
C
      QMIN=1.E+6
      QMAX=-1.E+6
C
      XMIN=1.E+6
      XMAX=1.E-6
C
      YMIN=1.E+6
      YMAX=1.E-6
C
C  INITIALIZE LOCATION OF MAXIMUM VALUES OF RHO, P AND Q
C
      KR=0
      LR=0
C
      KP=0
      LP=0
C
      KQ=0
      LQ=0
C
C  FIND THE MINIMUM AND MAXIMUM VALUES OF RHO, P, Q, R AND Z
C
```

```
      DO 715 L=LMNP,LMX
C
      DO 714 K=KMNP,KMX
C
      IF(RHO(K,L).LE.RMAX)GO TO 701
      RMAX=RHO(K,L)
      KR=K
      LR=L
C
  701 CONTINUE
C
      IF(P(K,L).LE.PMAX)GO TO 702
      PMAX=P(K,L)
      KP=K
      LP=L
C
  702 CONTINUE
C
      IF(Q(K,L).LE.QMAX)GO TO 703
      QMAX=Q(K,L)
      KQ=K
      LQ=L
C
  703 CONTINUE
C
      RMIN=AMIN1(RMIN,RHO(K,L))
      PMIN=AMIN1(PMIN,P(K,L))
      QMIN=AMIN1(QMIN,Q(K,L))
C
      XMIN=AMIN1(XMIN,R(K,L))
      XMAX=AMAX1(XMAX,R(K,L))
C
      YMIN=AMIN1(YMIN,Z(K,L))
      YMAX=AMAX1(YMAX,Z(K,L))
C
  714 CONTINUE
C
  715 CONTINUE
C
C PRINT PROBLEM PARAMETERS
C
      WRITE(NO,717) NYCL,TNUP,DTNPH,DTN,VERSION,WHER,WHEN
  717 FORMAT(6H NYCL ,I6,6H TIME ,E12.4,7H DTNPH ,E12.4,5H DTN ,
     X E12.4,9H VERSION ,F4.1,2A10)
C
      WRITE(NO,718) PMAX,KP,LP,QMAX,KQ,LQ,RMAX,KR,LR
  718 FORMAT(14H MAXIMUM (K,L),E12.4,2I4,3H P ,E12.4,2I4,3H Q ,
     X E12.4,2I4,5H RHO )
      UVTEST=1.E-5
      KL=KMN
      LL=LMN
      KU=KMX
      LU=LMX
      IF(IGEN.EQ.0) GO TO 720
       UVTEST=-100.
C PRINT ALL MESH POINTS
C IGEN.NE.0 WILL RESULT IN EDIT OF ENTIRE MESH,=0 ONLY ACTIVE ZONES
      KL=KMN-1
      LL=LMN-1
      KU=KMX+1
      LU=LMX+1
 720  CONTINUE
C BEGIN EDIT
      DO 740 L=LL,LU
      WRITE(NO,725)
  725 FORMAT(8H   L   K,4X,1HR,10X,1HZ,10X,1HU,10X,1HW,10X,3HRHO,
     X       8X,1HE,10X,1HP,10X,1HQ,10X,2HAJ,9X,5HTHETA)
      DO 738 K=KL,KU
      IF((ABS(U(K,L))+ABS(W(K,L))).LE.UVTEST)GO TO 738
C DONT PRINT VARIABLES IF NO MOTION
      WRITE(NO,726)L,K,R(K,L),Z(K,L),U(K,L),W(K,L),RHO(K,L),E(K,L)
     X,P(K,L),Q(K,L),AJ(K,L),TEMP(K,L)
 726  FORMAT(2I4,10E11.3)
 738  CONTINUE
 740  CONTINUE
C
      NET(NED)=NECOND(I1)
C
      NPT(NED)=NECOND(I1)
      NED=NED+1
      IF(NED.GT.20) NED=1
C
      RETURN
      END
      SUBROUTINE TEMPCAL ( RHO , EPS
C
      COMMON /EOSCOM/ KEOS,TARG1,TARG2,TARG3,RARG1,RARG2,RARG3,
     X  FUNC1,FUNC2,FUNC3,TEMPS,EPS,EPSO
C
C INVERSE TABLE LOOK-UP
C
      DATA P1M6/1.E-6/
      TARG1=0.
C
      CALL IES2
C E=EOS(0,RHO)
      EPSO=FUNC1
      TEMPS=0.
      IF(EPS.LT.EPSO) RETURN
C RETURN TEMPETA = 0 IF BELOW TABLE
      TEMPS=10.*EPS
C INITIAL GUESS
   10 TARG1=TEMPS
C
      CALL IES2
C
      FUNC2=FUNC1
      TARG1=TARG1+P1M6
C
      CALL IES2
C
      DTEMP=P1M6*((EPS-FUNC2)/(FUNC1-FUNC2))
      TEMPS=TEMPS+DTEMP
      IF(TEMPS.LT.P1M6) GO TO 20
      IF(ABS(DTEMP).GT.P1M6) GO TO 10
CONVERGED
      RETURN
   20 TEMPS=0.
      RETURN
      END
      SUBROUTINE JES
C
      COMMON /EOSCOM/ KEOS,TARG1,TARG2,TARG3,RARG1,RARG2,RARG3,
     X  FUNC1,FUNC2,FUNC3,TEMPS,EPS,EPSO
C
      COMMON /COM2/ NTSV(2),NRSV(2),MSV(2),TES(7),RES(9)
     X ,AES(12),BES(12),CES(12),DES(12),EES(12),FES(12),GES(12)
     X ,HES(12),PES(12),ITES(3),IRES(3),IZES(3)
C
      N=1
      RETURN
      ENTRY IES1
      N=1
      EXTT=1.
      EXTR=1.
      TARG=TARG1
      RARG=RARG1
      IBOUND=0
      IESTB=1
      GO TO 5000
  110 CONTINUE
      FUNC = AES(M)+RARG*(BES(M)+RARG*DES(M))
     1 +TARG*(CES(M)+RARG*(FES(M)+RARG*GES(M))
     2 +TARG*(EES(M)+RARG*(HES(M)+RARG*PES(M))))
      FUNC1=FUNC*EXTT*EXTR
      RETURN
C IES2 ENERGY=FUNCTION(TEMPETA RHO)
C
      ENTRY IES2
      N=2
      EXTT=1.
      TARG=TARG1
      RARG=RARG1
      IBOUND=0
      IESTB=2
      GO TO 5000
  210 CONTINUE
      FUNC = AES(M)+RARG*(BES(M)+RARG*DES(M))
     1 +TARG*(CES(M)+RARG*(FES(M)+RARG*GES(M))
     2 +TARG*(EES(M)+RARG*(HES(M)+RARG*PES(M))))
      FUNC1=FUNC*EXTT
      RETURN
C                    TABLE LOOK UP
C
 5000 NT=NTSV(N)
      NR=NRSV(N)
      MLR = 0
      MLT = 0
C
      IF(TES(NT).GT.TARG)   GO TO 5100
      IF(TES(NT+1).LE.TARG)  GO TO 5200
C
C                    TARG IN SAME T STRIP AS FOR PREVIOUS ENTRY
C
      IF(RES(NR).GT.RARG)    GO TO 5300
      IF(RES(NR+1).LE.RARG) GO TO 5400
C
C                    TARG AND RARG IN SAME BOX AS FOR PREVIOUS ENTRY
C                    M SAME AS FOR PREVIOUS ENTRY,FAST RETURN
C
```

```
      M=MSV(N)
      GO TO (110,210) ,IESTB
C
C                    T SEARCH
C
C                    TARG BELOW T STRIP OF PREVIOUS ENTRY
C
C                    OUT OF TABLE TEST, LOW T
C
 5100 IF(NT.LE.ITES(N)) GO TO 5115
C
C                    SEARCH TO NEXT LOWER T STRIP
C
 5105 NT=NT-1
      IF(TES(NT).GT.TARG) GO TO 5120
C
C                    STRIP CONTAINING TARG FOUND, BEGIN R SEARCH
C
      IF(RES(NR)-RARG) 5410,5310,5320
C
C                    TARG BELOW LOWEST TABLE ARGUMENT AND
C                    WAS BELOW TEMPAT ARGUMENT ON PREVIOUS ENTRY
C
 5115 MLT=-1
      EXTT=EXTT*TARG/TES(NT)
      TARG=TES(NT)
C
      IF(RES(NR).GT.RARG) GO TO 5300
      IF(RES (NR+1).LE.RARG) GO TO 5400
      M = MSV(N)
      GO TO (110,210) ,IESTB
C
C                    OUT OF TABLE TEST, LOW T
C
 5120 IF(NT.GT.ITES(N)) GO TO 5105
C
C                    TARG BELOW LOWEST TABLE ARGUMENT BUT
C                    WAS NOT BELOW TEMPAT ARGUMENT ON PREVIOUS ENTRY
C
      MLT=-1
      EXTT=EXTT*TARG/TES(NT)
      TARG=TES(NT)
C
C                    BEGIN R SEARCH
C
      IF(RES(NR)-RARG)    5410,5310,5320
C
C                    OUT OF TABLE TEST, HIGH T
C
 5200 IF(NT-ITES(N+1)+2) 5205,5215,5205
C
C                    SEARCH TO NEXT HIGHER T STRIP
C
 5205 NT=NT+1
      IF(TES(NT+1).LE.TARG) GO TO 5220
C
C                    STRIP CONTAINING TARG FOUND, BEGIN R SEARCH
C
      IF(RES(NR)-RARG)    5410,5310,5320
C
```

```
C                         TARG ABOVE HIGHEST TABLE ARGUMENT AND
C                         WAS ABOVE TEMPAT ARGUMENT ON PREVIOUS ENTRY
C
 5215 MLT=1
      EXTT=EXTT*TARG/TES(NT+1)
      TARG=TES(NT+1)
C
      IF(RES(NR).GT.RARG) GO TO 5300
      IF(RES(NR+1).LE.RARG) GO TO 5400
      M = MSV(N)
      GO TO (110,210) ,IESTB
C
C                         OUT OF TABLE TEST, HIGH T
C
 5220 IF(NT-ITES(N+1)+2) 5205,713,5205
C
C                         TARG ABOVE HIGHEST TABLE ARGUMENT BUT WAS
C                         NOT ABOVE TEMPAT ARGUMENT ON PREVIOUS ENTRY
C
 713  MLT=1
      EXTT=EXTT*TARG/TES(NT+1)
      TARG=TES(NT+1)
C
C                         BEGIN R SEARCH
C
      IF(RES(NR)-RARG)   5410,5310,5320
C
C                         OUT OF TABLE TEST, LOW R
C
 5320 IF(NR.GT.IRES(N)) GO TO 5305
C
C                         RARG BELOW LOWEST TABLE ARGUMENT BUT WAS
C                         NOT BELOW TEMPAT ARGUMENT ON PREVIOUS ENTRY
C
      MLR=-1
      EXTR=EXTR*RARG/RES(NR)
      RARG=RES(NR)
      GO TO 5310
C
C                         R SEARCH
C                         RARG BELOW R STRIP OF PREVIOUS ENTRY
C                         OUT OF TABLE TEST, LOW R
C
 5300 IF(NR.GT.IRES(N)) GO TO 5305
C
C                         RARG BELOW LOWEST TABLE ARGUMENT AND
C                         WAS BELOW TEMPAT ARGUMENT ON PREVIOUS ENTRY
C
      MLR=-1
      EXTR=EXTR*RARG/RES(NR)
      RARG=RES(NR)
      M = MSV(N)
      GO TO (110,210) ,IESTB
C
C                         SEARCH TO NEXT LOWER R STRIP
C
 5305 NR=NR-1
      IF(RES(NR) - RARG) 5310,5310,5320
C
C                         BOX CONTAINING TARG AND RARG FOUND, COMPUTE NEW M
```

```
C
 5310 M=IZES(N)+(ITES(N+1)-ITES(N)-1)*(NR-IRES(N))+NT-ITES(N)
      NTSV(N)=NT
      NRSV(N)=NR
      MSV(N)=M
      GO TO (110,210) ,IESTB
C
C                  OUT OF TABLE TEST, HIGH R
C
 5400 IF(NR - IRES(N+1)+2) 5405,5415,5405
C
C                  SEARCH TO NEXT HIGHER R STRIP
C
 5405 NR=NR+1
 5410 IF(RES(NR+1).GT.RARG) GO TO 5310
      IF(NR-IRES(N+1)+3) 5405,5405,719
C
C                  RARG ABOVE HIGHEST TABLE ARGUMENT BUT WAS
C                  NOT  ABOVE TEMPAT ARGUMENT ON PREVIOUS ENTRY
C
 719  MLR=1
      EXTR=EXTR*RARG/RES(NR+1)
      RARG=RES(NR+1)
      GO TO 5310
C
C                  RARG ABOVE HIGHEST TABLE ARGUMENT BUT
C                  M SAME AS ON PREVIOUS ENTRY
C
 5415 MLR=1
      EXTR=EXTR*RARG/RES(NR+1)
      RARG=RES(NR+1)
      M = MSV(N)
      GO TO (110,210) ,IESTB
      END
      SUBROUTINE SETUP
C
      COMMON /COM2/ NTSV(2),NRSV(2),MSV(2),TES(7),RES(9)
     X ,AES(12),BES(12),CES(12),DES(12),EES(12),FES(12),GES(12)
     X ,HES(12),PES(12),ITES(3),IRES(3),IZES(3)
C
      CALL JES
C DEFINE A GAMMA LAW GAS EQUATION OF STATE FOR BIQUAD ROUTINE
      NTSV(1)=1
      NRSV(1)=1
      MSV(1)=1
      NTSV(2)=4
      NRSV(2)=5
      MSV(2)=7
      ITES(1)=1
      IRES(1)=1
      IZES(1)=1
      ITES(2)=4
      IRES(2)=5
      IZES(2)=7
      ITES(3)=7
      IRES(3)=9
      IZES(3)=13
       TES(   1) =        .0E+00
       TES(   2) =   1.0000E+00
       TES(   3) =   1.0000E+02
       TES(   4) =        .0E+00
       TES(   5) =   1.0000E+00
       TES(   6) =   1.0000E+02
       TES(   7) =        .0E+00
       RES(   1) =        .0E+00
       RES(   2) =   3.0000E+00
       RES(   3) =   3.0000E+02
       RES(   4) =   3.0000E+10
       RES(   5) =        .0E+00
       RES(   6) =   3.0000E+00
       RES(   7) =   3.0000E+02
       RES(   8) =   3.0000E+10
       RES(   9) =        .0E+00
       AES(   1) =        .0E+00
       BES(   1) =        .0E+00
       CES(   1) =        .0E+00
       DES(   1) =        .0E+00
       EES(   1) =        .0E+00
       FES(   1) =   6.6667E-02
       GES(   1) =  -1.2953E-16
       HES(   1) =  -4.4409E-16
       PES(   1) =  -9.2519E-17
       AES(   2) =        .0E+00
       BES(   2) =  -4.7184E-16
       CES(   2) =        .0E+00
       DES(   2) =  -1.7146E-16
       EES(   2) =        .0E+00
       FES(   2) =   6.6667E-02
       GES(   2) =  -4.8247E-17
       HES(   2) =   1.0408E-17
       PES(   2) =  -2.3426E-18
       AES(   3) =        .0E+00
       BES(   3) =        .0E+00
       CES(   3) =  -8.0183E-17
```

```
1446         DES(   3) =        .0E+00
1447         EES(   3) =  -5.8595E-16
1448         FES(   3) =   6.6667E-02
1449         GES(   3) =   2.2166E-18
1450         HES(   3) =   4.4409E-16
1451         PES(   3) =  -4.8187E-19
1452         AES(   4) =  -1.9697E-15
1453         BES(   4) =  -1.1102E-15
1454         CES(   4) =   1.3180E-15
1455         DES(   4) =   3.5824E-19
1456         EES(   4) =  -1.4404E-17
1457         FES(   4) =   6.6667E-02
1458         GES(   4) =   1.3792E-18
1459         HES(   4) =  -3.4694E-18
1460         PES(   4) =  -2.7257E-21
1461         AES(   5) =        .0E+00
1462         BES(   5) =        .0E+00
1463         CES(   5) =  -9.4344E-14
1464         DES(   5) =        .0E+00
1465         EES(   5) =  -3.3553E-14
1466         FES(   5) =   6.6667E-02
1467         GES(   5) =        .0E+00
1468         HES(   5) =        .0E+00
1469         PES(   5) =        .0E+00
1470         AES(   6) =  -4.5296E-14
1471         BES(   6) =  -1.0547E-15
1472         CES(   6) =  -8.3441E-14
1473         DES(   6) =   1.8567E-27
1474         EES(   6) =   8.3693E-16
1475         FES(   6) =   6.6667E-02
1476         GES(   6) =  -2.0424E-27
1477         HES(   6) =        .0E+00
1478         PES(   6) =   1.8567E-28
1479         AES(   7) =        .0E+00
1480         BES(   7) =        .0E+00
1481         CES(   7) =   1.0000E-01
1482         DES(   7) =        .0E+00
1483         EES(   7) =        .0E+00
1484         FES(   7) =   2.0362E-15
1485         GES(   7) =  -5.[illegible]E-16
1486         HES(   7) =   9.7[illegible]E-16
1487         PES(   7) =  -1.2[illegible]E-16
1488         AES(   8) =  -1.7[illegible]E-15
1489         BES(   8) =   1.[illegible]E-17
1490         CES(   8) =   1.0000E-01
1491         DES(   8) =  -2.0[illegible]E-17
1492         EES(   8) =        .0E+00
1493         FES(   8) =   2.9993E-15
1494         GES(   8) =  -6.4311E-16
1495         HES(   8) =   6.5417E-20
1496         PES(   8) =  -2.0933E-18
1497         AES(   9) =        .0E+00
1498         BES(   9) =        .0E+00
1499         CES(   9) =   1.0000E-01
1500         DES(   9) =        .0E+00
1501         EES(   9) =   4.4409E-16
1502         FES(   9) =  -2.6587E-17
1503         GES(   9) =  -6.0233E-21
1504         HES(   9) =   1.9214E-17
1505         PES(   9) =   6.0233E-20
```

```
      AES( 10) = -1.8874E-15
      BES( 10) =  1.1273E-17
      CES( 10) =  1.0000E-01
      DES( 10) =  1.2168E-20
      EES( 10) = -6.9389E-18
      FES( 10) = -1.8510E-17
      GES( 10) =  4.2030E-20
      HES( 10) = -1.3521E-19
      PES( 10) =  1.2168E-23
      AES( 11) =         .0E+00
      BES( 11) =         .0E+00
      CES( 11) =  1.0000E-01
      DES( 11) =         .0E+00
      EES( 11) =  8.8818E-16
      FES( 11) = -5.8341E-26
      GES( 11) = -6.3947E-36
      HES( 11) =  5.8341E-26
      PES( 11) = -1.8808E-36
      AES( 12) = -1.9429E-15
      BES( 12) = -8.9878E-26
      CES( 12) =  1.0000E-01
      DES( 12) = -5.3701E-36
      EES( 12) = -6.9389E-18
      FES( 12) =  8.9962E-26
      GES( 12) = -2.8867E-36
      HES( 12) = -8.4134E-29
      PES( 12) = -1.8745E-38
      RETURN
      END
      SUBROUTINE PROJCT
C
C  THIS SUBROUTINE REFLECTS AN INTERIOR POINT ACROSS THE BOUNDARY
C
      COMMON /PROGG/RO,ZO,R1,Z1,RP,ZP,RR,ZZ
C
C  REFLECT (RP,ZP) TO (RR,ZZ)
C  WHERE (RO,ZO) AND (R1,Z1) ARE BOUNDARY POINTS
C
      WW=(2.*(Z1-ZO))/((R1-RO)**2+(Z1-ZO)**2)
      ALP=1.-(Z1-ZO)*WW
      BET=(R1-RO)*WW
      RR=RO+(RP-RO)*ALP + (ZP-ZO)*BET
      ZZ=ZO+(RP-RO)*BET - (ZP-ZO)*ALP
C
      RETURN
      END
      SUBROUTINE CONDUCT
C
      COMMON /KLS/ K,L,DEBUG,VERSION,WHER,WHEN,P1D6,PIE,IGEN,P1D2
     X ,DTC,KC,LC,DTEN,KEN,LEN,SKE,HN,SIEL,CNN,ENC,ENH,ENCG,WN
     X  ,NCP
C
      COMMON /COMN/  R(33,33),Z(33,33),U(33,33),RHO(33,33),Q(33,33)
     X            ,E(33,33),P(33,33),AJ(33,33),S(33,33),NBC(33,33)
     X        ,W(33,33),TEMP(33,33)
     X , A(33,33),B(33,33),CC(33,33),DUM(33,33),CBB(33,33)
     X , DBB(33,33),CAP(33,33),SIG(33,33),TS(33,33)
C
      COMMON /PARAM/ NYCL,TNUP,DTNUP,DTN,DTNPH,DTNMH,EDTIME,EDDT
     X       ,GAM,GAMZ,COF,C1F,C1,TMAX,DTMAX,DTMIN,TFLR,NOHYD
     X       ,C2,P2,P3,NO,NTTY,NED
C
      COMMON /KLSPACE/ KMN,LMN,KMX,LMX,KMXZ,LMXZ,KMNP,LMNP,KMXP,LMXP
C
C     COMMON /EOSCOM/ KEOS,TARG1,TARG2,TARG3,RARG1,RARG2,RARG3,
C    X  FUNC1,FUNC2,FUNC3,TEMPS,EPS,EPSO
C
C ELECTRON CONDUCTION -LU-
C
      DO 10 L=LMN,LMX
      DO 10 K=KMN,KMX
      CAP(K,L)=.1
      CC(K,L)=(.0001*SQRT(TEMP(K,L))*TEMP(K,L)**2)/AJ(K,L)
      SIG(K,L)=DUM(K,L)*CAP(K,L)/DTNPH
      TS(K,L)=TEMP(K,L)
 10   CONTINUE
C
      DO 12 L=LMN,LMX
       DO 12 K=KMN,KMXZ
      CBB(K,L)=(2.*CC(K+1,L)*CC(K+1,L+1))/(CC(K+1,L)+CC(K+1,L+1))
     X * (.5*(R(K,L)+R(K+1,L))*((R(K+1,L)-R(K,L))**2
     X      +(Z(K+1,L)-Z(K,L))**2) )
  12  CONTINUE
       DO 14 L=LMN,LMXZ
       DO 14 K=KMN,KMX
      DBB(K,L)=(2.*CC(K+1,L+1)*CC(K,L+1))/(CC(K+1,L+1)+CC(K,L+1))
     X * (.5*(R(K,L)+R(K,L+1))*((R(K,L+1)-R(K,L))**2
     X      +(Z(K,L+1)-Z(K,L))**2) )
  14  CONTINUE
C
C BOUNDARY CONDITIONS
      DO 17 L=LMN,LMX
      A(KMN,L)=0.
      B(KMN,L)=TEMP(KMN,L)
      DBB(KMN,L)=0.
  17  CONTINUE
C
      DO 19 K=KMN,KMX
      A(K,LMN)=0.
      B(K,LMN)=TEMP(K,LMN)
      CBB(K,LMX)=0.
      CBB(K,LMN)=0.
  19  CONTINUE
C
C
C
```

```
C.......... Z SWEEP
C
       DO   53 K=KMNP,KMX
       DO   51 L=LMNP,LMX
       DUM(K,L)=SIG(K,L)+CBB(K-1,L)+CBB(K-1,L-1)*(1.-A(K,L-1))
       A(K,L)=CBB(K-1,L)/DUM(K,L)
       B(K,L)=(SIG(K,L)*TEMP(K,L)+CBB(K-1,L-1)*B(K,L-1)
     X )/DUM(K,L)
   51 CONTINUE
C...... ALPHA,BETA FORWARD
      ML=LMX+1
      DO 52 L=LMNP,LMX
      ML=ML-1
      TEMP(K,ML)=A(K,ML)*TEMP(K,ML+1)+B(K,ML)
   52 CONTINUE
C BACK SUBSTITUTION
  53  CONTINUE
C
C.......... Z SWEEP END
C
C.......... R SWEEP
C
       DO   43 L=LMNP,LMX
       DO   41 K=KMNP,KMX
       DUM(K,L)=SIG(K,L)+DBB(K,L-1)+DBB(K-1,L-1)*(1.-A(K-1,L))
       A(K,L)=DBB(K,L-1)/DUM(K,L)
      B(K,L)=(SIG(K,L)*TEMP(K,L)+DBB(K-1,L-1)*B(K-1,L)
     X )/DUM(K,L)
   41 CONTINUE
C...... ALPHA BETA FORWARD SWEEP
      ML=KMX+1
      DO 42 K=KMNP,KMX
      ML=ML-1
      TEMP(ML,L)=A(ML,L)*TEMP(ML+1,L)+B(ML,L)
  42  CONTINUE
C BACK SUBSTITUTION R DIRECTION
  43  CONTINUE
C
C.......... R SWEEP END
C COMPUTE DT CONTROL FOR HEAT CONDUCTION
C
      YE=0.
      KEN=0
      LEN=0
      DO 111 L=LMNP,LMX
      DO 111 K=KMNP,KMX
C GET NEW ENERGY
      ENH=ENH+E(K,L)*RHO(K,L)*S(K,L)
C
      TARG1=TEMP(K,L)
      RARG1=RHO(K,L)
      CALL IES2
C
      E(K,L)=AMAX1(FUNC1,1.E-30)
      ENC=ENC+E(K,L)*RHO(K,L)*S(K,L)
      IF(TS(K,L).EQ.0.) GO TO 109
      TEMPR=ABS((TEMP(K,L)-TS(K,L))/TS(K,L))
      IF(TEMPR.LE.YE) GO TO 109
      YE=TEMPR
      KEN=K
      LEN=L
  109 TEMP(K,L)=TS(K,L)
  111 CONTINUE
      IF(YE.EQ.0.) GO TO 118
      DTEN=(.1*DTNPH)/YE
  118 CONTINUE
C ENERGY BALANCE HN
      DO 122 K=2,KMX
      HN=HN+DTNPH*CBB(K-1,LMN)*(TEMP(K,LMN)-TEMP(K,LMN+1))
     X    +DTNPH*CBB(K-1,LMX)*(TEMP(K,LMX+1)-TEMP(K,LMX))
  122 CONTINUE
C
      DO 124 L=2,LMX
      HN=HN+DTNPH*DBB(KMN,L-1)*(TEMP(KMN,L)-TEMP(KMN+1,L))
     X    +DTNPH*DBB(KMX,L-1)*(TEMP(KMX+1,L)-TEMP(KMX,L))
  124 CONTINUE
C
      RETURN
      END
      SUBROUTINE HWORK
C
      COMMON /KLS/ K,L,DEBUG,VERSION,WHER,WHEN,P1D6,PIE,IGEN,P1D2
     X ,DTC,KC,LC,DTEN,KEN,LEN,SKE,HN,SIEL,CNN,ENC,ENH,ENCG,WN
     X  ,NCP
C
      COMMON /COMN/  R(33,33),Z(33,33),U(33,33),RHO(33,33),Q(33,33)
     X           ,E(33,33),P(33,33),AJ(33,33),S(33,33),NBC(33,33)
     X       ,W(33,33),TEMP(33,33)
     X ,  A(33,33),B(33,33),CC(33,33),DUM(33,33),CBB(33,33)
     X ,  DBB(33,33),CAP(33,33),SIG(33,33),TS(33,33)
C
      COMMON /PARAM/ NYCL,TNUP,DTNUP,DTN,DTNPH,DTNMH,EDTIME,EDDT
     X      ,GAM,GAMZ,COF,C1F,C1,TMAX,DTMAX,DTMIN,TFLR,NOHYD
     X      ,C2,P2,P3,NO,NTTY,NED
C
      COMMON /KLSPACE/ KMN,LMN,KMX,LMX,KMXZ,LMXZ,KMNP,LMNP,KMXP,LMXP
C
C SUM THE HYDRO WORK ON THE BOUNDARY
C
      Z1=DTNPH/8.
C
      DO 510 K=KMNP,KMX
C
      WN=WN+Z1*(P(K,LMN+1)+P(K,LMN)+Q(K,LMN+1)+Q(K,LMN))
     X  *( (U(K,LMN)+U(K-1,LMN))*(Z(K,LMN)-Z(K-1,LMN))
     X    -(W(K,LMN)+W(K-1,LMN))*(R(K,LMN)-R(K-1,LMN))
     X   )*(R(K,LMN)+R(K-1,LMN))
C
      WN=WN-Z1*(P(K,LMX+1)+P(K,LMX)+Q(K,LMX+1)+Q(K,LMX))
     X  *( (U(K,LMX)+U(K-1,LMX))*(Z(K,LMX)-Z(K-1,LMX))
     X    -(W(K,LMX)+W(K-1,LMX))*(R(K,LMX)-R(K-1,LMX))
     X   )*(R(K,LMX)+R(K-1,LMX))
C
  510  CONTINUE
C
      DO 515 L=LMNP,LMX
C
      WN=WN+Z1*(P(KMN+1,L)+P(KMN,L)+Q(KMN+1,L)+Q(KMN,L))
     X  *( (U(KMN,L)+U(KMN,L-1))*(Z(KMN,L)-Z(KMN,L-1))
     X    -(W(KMN,L)+W(KMN,L-1))*(R(KMN,L)-R(KMN,L-1))
     X   )*(R(KMN,L)+R(KMN,L-1))
C
      WN=WN-Z1*(P(KMX+1,L)+P(KMX,L)+Q(KMX+1,L)+Q(KMX,L))
     X  *( (U(KMX,L)+U(KMX,L-1))*(Z(KMX,L)-Z(KMX,L-1))
     X    -(W(KMX,L)+W(KMX,L-1))*(R(KMX,L)-R(KMX,L-1))
     X   )*(R(KMX,L)+R(KMX,L-1))
C
  515  CONTINUE
C
      RETURN
      END
      FUNCTION NECOND(IARG)
      IAG=0
      AA1=SECOND(IAG)
      NECOND=(AA1-AA2)*1.E+6
      AA2=AA1
      RETURN
      END
```