LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Design Considerations for a Partial Differential Equation Machine

Computation Structures Group Memo 178
January 1980

Arvind
Randal E. Bryant

This memo was presented as a paper at the Scientific Computer Information Exchange (SCIE) Meeting at Lawrence Livermore Laboratory, Livermore, CA, Sept. 12-13, 1979.

This research was supported by the National Science Foundation under grant MCS-7902782, and by the University of California, Lawrence Livermore Laboratory, under contract 8545403.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

CAMBRIDGE, MASSACHUSETTS 02139

Integrated Circuit Memo No. 80-3

January 4, 1980

DESIGN CONSIDERATIONS FOR A PARTIAL DIFFERENTIAL EQUATION MACHINE*

by

Arvind†

Randal E. Bryant‡

ABSTRACT

Partial differential equation (PDE) simulation provides an attractive area for the application of highly parallel computer systems. The regular and static structures of these problems and the limited data dependencies allow them to be mapped onto a system consisting of many interconnected processors. This paper presents an analysis of a program for simulating the hydrodynamic motion and heat flow in a compressible fluid. Based on this analysis, some of the issues in designing programming languages and computer architectures for PDE simulations are discussed. The data flow model of computation is seen to provide an attractive means for managing the complexity of highly parallel systems. Data flow concepts can be applied to relatively simple architectures specifically designed for PDE simulation.

---

* This research was supported by the National Science Foundation under grant MCS-7902782, and by the University of California, Lawrence Livermore Laboratory, under contract no. 8545403. This memo was presented as a paper at the Scientific Computer Information Exchange (SCIE) Meeting at Lawrence Livermore Laboratory, Livermore, CA, Sept. 12-13, 1979.

† Laboratory for Computer Science, and Department of Electrical Engineering and Computer Science, M.I.T., Room NE43-535; (617) 253-6090.

‡ Laboratory for Computer Science, and Department of Electrical Engineering and Computer Science, M.I.T., Room NE43-531; (617) 253-6212.

Design Considerations for a Partial Differential Equation Machine[1]

Arvind, and Randal E. Bryant
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract

Partial differential equation (PDE) simulation provides an attractive area for the application of highly parallel computer systems. The regular and static structures of these problems and the limited data dependencies allow them to be mapped onto a system consisting of many interconnected processors. This paper presents an analysis of a program for simulating the hydrodynamic motion and heat flow in a compressible fluid. Based on this analysis, some of the issues in designing programming languages and computer architectures for PDE simulations are discussed. The data flow model of computation is seen to provide an attractive means for managing the complexity of highly parallel systems. Data flow concepts can be applied to relatively simple architectures specifically designed for PDE simulation.

Introduction

Partial differential equation (PDE) simulation has often been proposed as an ideal area for the application of highly concurrent computer architectures. The high computational requirements of these problems provide an incentive for high speed computation, while the regularity and minimal data dependencies provide hope that this speed can be achieved through parallelism.

Highly parallel computer architectures diverge from traditional, sequential computers to different degrees and in a variety of different ways. This paper examines how a computer architecture and high level programming language can be developed to achieve high performance at a reasonable cost, while maintaining programmability. Some of the architectural considerations include: how the processing resources are allocated, how the activities of the processors are synchronized, and what forms of communication are allowed between processors. Other potentially important decisions such as mechanisms for achieving fault tolerance and for input and output will not be considered.

(1) This research was supported by the National Science Foundation under grant MCS-7902782, and by the University of California, Lawrence Livermore Laboratory under contract no. 8545403.

While the above-mentioned design issues are directed toward the computer architecture, they will also strongly influence the design of the programming languages supported by the architecture. To provide reasonable programmability, the architecture must support some abstract model of computation which can form a basis for a high level programming language. For example, traditional architectures can be viewed as performing a sequence of updates to a set of memory cells, forming the basis for languages such as FORTRAN. Highly parallel architectures, however, must diverge from this model and hence will require new forms of programming languages. Thus we will discuss computer architectures and the languages for these architectures together.

We will assume the system consists of a number of processing elements (or simply "processors"), each capable of storing and executing a program and of storing data. Examples of such systems include the Irvine data flow architecture [3, 8], and the Utah data flow architectures [5, 10]. This model does not encompass the MIT data flow architecture [7] in which the functions of program storage, instruction execution, and data storage are performed by separate units. Nonetheless, much of the analysis should apply to this system as well.

The SIMPLE Code

As a focus for the study we have been studying the SIMPLE code [4], a 1500 line FORTRAN program developed at Lawrence Livermore Laboratories. The SIMPLE code is a simplified version of a program for simulating both the *hydrodynamics*, or mechanical motion, and the *heat flow*, or the conduction of heat between regions of a compressible fluid. Most of the simplifications serve only to decrease the total size of the program without decreasing the complexities of the numerical model. In comparison to other PDE simulation programs, such as for weather simulation or aerodynamic modeling, this program simulates systems undergoing very rapid changes with extremes of temperature and pressure and also with many shocks. As a result this simulation requires a more complex numerical model. The SIMPLE code may present somewhat of a "worst case" example in terms of potential concurrency and regularity of computation.

Although mechanical motion and heat conduction proceed simultaneously in the physical system, SIMPLE separates the two during each time step, simulating first the hydrodynamics and then the heat flow. The fluid is represented in a two-dimensional, Lagrangian formulation. A block diagram for the program is shown in Figure 1. During the hydrodynamics phase of a cycle, the program uses the positions x, and velocities v of the node points and the pressures p, artificial viscosities [1] q, and densities ρ of the zones to compute new positions x' of the nodes by an explicit difference method. Then new values for density ρ' and artificial viscosity q' are calculated along with intermediate values of energy $\hat{\epsilon}$. The heat conduction phase takes these intermediate energy values and transfers energy between zones to represent the flow of heat resulting in new energies ϵ' by an alternating-direction implicit difference method. It also computes a new set of zone pressures p' based on the energy. Finally, a value for the size of the next time step Δt is calculated. The time step must be kept small enough to maintain the stability of the computation [11]. This requires calculating the allowable time step for each zone and finding the minimum of these values over the entire mesh. Following the time step calculation a new cycle can begin.

(1) Artificial viscosity [11] is a computational technique used to smooth out shocks

Inherent Parallelism and Computational Requirements

An analysis of the SIMPLE program reveals the quantity of computation required and the forms of parallelism allowed for a typical PDE simulation. In SIMPLE the amount of concurrency and the data dependencies vary greatly in the different phases of the computation, because of the different numerical methods used. These data dependencies have important implications for exploiting the potential concurrency of the program.

Figure 2 shows the partial ordering on the program variables imposed by the data dependencies. This diagram omits those arcs implied by transitivity. As can be seen, the data dependencies impose a nearly linear ordering on the computations. Most of the variables, however, are two-dimensional arrays. If we consider the array elements as individual values to be computed we can study their data dependencies as well. Figure 2 shows four classes of dependencies:

local:	array element (k,l) depends only on elements (k,l) of the other arrays.
neighbor:	array element (k,l) depends on elements (k,l), $(k+1,l)$, $(k-1,l)$, $(k,l+1)$, $(k,l-1)$.
global:	a scalar value depends on all elements of the arrays.
scalar:	every array element depends on some scalar value.

As can be seen, most zone and node computations depend only on values from neighboring nodes and zones. In fact, many computations are fully localized. In only a few cases must the results of one computation be received from the neighbors before another computation can proceed. This does not take into account any sharing of program or constant data between zone computations to reduce the total storage requirements.

Figure 3 depicts the potential concurrency and computational requirements graphically for a 100 by 100 zone mesh assuming that the two equation of state calculations for each zone take

two iterations on average to converge. This figure shows how the computation for one time step would proceed if unlimited processing and communication resources were available. The abscissa shows the elapsed time in units of floating point operation times (all operations are assumed to require the same time.) The ordinate shows the total number of operations proceeding concurrently, typically a small constant times the number of concurrent zone computations. The area of each shaded region then shows the total number of operations for each section of the program.

As Figure 3 demonstrates, with unbounded processing capability the heat conduction section would require 86% of the elapsed time, even though it represents only 5% of the total number of operations due to the restricted concurrency of this section. This analysis is somewhat misleading, however, because even the heat conduction section would allow approximately 220 operations to proceed concurrently. While this is substantially less than the 24,000 to 48,000 concurrent operations allowed by other sections of the program, it still exceeds the capacity of any existing concurrent architecture. The desire for higher concurrency may ultimately call for a different numerical method, but this conclusion should not be reached too hastily. Figure 3 also does not show the possible overlapping of calculations for two time steps. In SIMPLE this possible is limited, because the Δt calculation requires the results from one time step before allowing the next time step to begin.

Irregularities in the Computation

In most sections of SIMPLE, an identical set of operations is performed for every zone. These sections could be carried out by a set of processors executing identical, or at least very similar, instruction streams. Certain aspects of the program, however, perturb this regularity, requiring a different set of operations for some of the zones. Any programming language or computer architecture which cannot deal with these irregularities efficiently may exact a large penalty in programmability or performance.

Boundary calculations always cause irregularities in PDE simulations. SIMPLE only allows a limited class of time-invariant boundary conditions, and the boundaries must correspond to the edges of the rectangular state variable arrays. Nonetheless, these boundary calculations differ in their form and data dependencies from the calculations for internal zones and typically require more computation. In more complex programs, a variety of time-varying boundary conditions may be specified, and the boundaries may cause the logical representations of the state variables to have irregular perimeters and holes. Calculations for boundary conditions will prove the downfall of any language or architecture which requires an identical set of operations over an entire array or vector.

Any part of the program for which the flow of control depends on data-dependent decisions may also cause irregularities in the program. For example, in two sections of SIMPLE the root of an equation is computed iteratively for each zone. The number of iterations required for convergence will differ from zone to zone. Each iteration requires a significant amount of computation, causing large variations in the amount of computation per zone. Similarly, another section of the program approximates a function with a piecewise-polynomial curve. Computing this function first requires searching a table for the appropriate set of coefficients with a data-dependent search time. Finally, whenever an exceptional condition is encountered in the computation, such as a quantity exceeding some upper or lower limit, the program must take steps to correct this condition. Thus, the data-dependent decisions in the program can cause both small and large irregularities in the overall structure of the computation.

Programming Languages for PDE Simulation

Once the difference equations for a PDE simulation have been specified, their coding in a FORTRAN-like language proceeds without difficulty. The array data structures and DO loop control structures provide adequate expressive power for most applications. These programs, however, do not run efficiently on existing high performance computers such as the Star-100, Cray-1, or Illiac IV. The programs must be carefully hand coded (often in assembly language) and optimized before the potential of these machines can be realized. Smart compilers have failed to bridge the gap from traditional languages to high performance machines.

This disappointing performance of FORTRAN programs stems largely from a mismatch of language and high performance architectures. A

FORTRAN program specifies the computation in terms of a sequence of updates to individual memory locations. Array and pipeline computers, however, operate most efficiently when working with entire arrays or vectors. Thus, the compiler (usually augmented by a human) must try to combine and restructure sections of code to make full use of vector instructions. If vectors must be stored contiguously in the memory, further complications arise.

The difficulties in programming existing high performance machines is further compounded by their restrictive architectures. To support high level languages efficiently an architecture must lend itself to a process of abstraction in which the exact size, configuration, and speed of the hardware components are masked. The architecture must then have the flexibility to achieve reasonable performance even with less than optimal programs. Unless the architecture supports some abstract model consistently and efficiently, the programmer will be forced to resort to machine-level coding to take full advantage of the machine's power.

We believe the data flow model of computation [6] provides a suitable basis to be supported by highly concurrent architectures and upon which high level languages can be built. As a basis for high level language, the data flow model allows programs to be written which express the maximal concurrency allowed by an algorithm. Control is based solely on the availability of data rather than on the sequential ordering of program statements. Hence, only data dependencies constrain the program's concurrency.

The data flow model supports *functional* programming languages in which program statements define functions from the input operands to the output values. In a functional language a statement can be executed (i.e. the function evaluated) as soon as the input operands have been computed. Functional languages contrast with *imperative* languages in which each statement defines a command for altering some memory location, and statements must in general be executed sequentially. With imperative languages concurrency can be achieved only by removing the unnecessary sequencing constraints in the program, whereas such constraints never appear in functional programs. Functional languages which have been designed with the data flow model as their basis include Id [3] and Val [1, 2].

Functional programming languages have been stereotyped as amusing diversions for academicians rather than serious tools for expressing production scientific programs. The syntax and data structures of languages such as Lisp seem foreign to most scientific programmers. Such difficulties arise not from their functional nature but rather from the purposes these languages are intended to serve. We believe that functional languages for scientific programming can be developed which will actually simplify the task of coding and maintaining programs. Attempts at reprogramming SIMPLE in Irvine dataflow (Id) have proved quite successful.

Architectures for PDE Simulation

Some high performance computer architectures, such as the Cray-1, have achieved remarkable success while maintaining the basic single sequence control. Others, such as the Star-100 and Illiac IV have failed to live up to their expectations. While the success of the Cray-1 can be ascribed largely to the quality of its engineering, it also results from a greater tolerance of the irregularities in the program structure. The Illiac IV operates efficiently only when performing an identical operation over an entire array, while the Star operates efficiently only on long vectors. Sections of the program requiring scalar or short vector operations move at a much slower pace. As a result, programs must be painstakingly reworked to maximize their regularity, often to a greater extent than is called for by the algorithm. For example, the holes and irregular perimeter of the mesh may be filled with "null" zones to rectangularize the state variable descriptions. The Cray-1, on the other hand, achieves reasonable performance with scalar and short vector computations. As a result, it can tolerate partially vectorized programs. Nonetheless, it too requires careful optimization to achieve maximum performance.

All existing architectures have tried to achieve high performance by maximizing the regularity in the program and then exploiting the parallelism allowed by this regularity. This approach will always force the programmer to carefully think in terms of how the program fits onto the machine. This level of thinking requires machine-level coding to provide the necessary degree of control. Furthermore, many programs

simply do not lend themselves to highly regular structuring. Future architectural developments must follow a new path if they are to achieve significantly higher performance and programmability.

As we have seen, PDE simulation programs potentially allow a high degree of concurrency in their execution. To exploit this concurrency effectively, a computer must be capable of concurrently executing different instructions on different data. Within this framework, one can choose from a variety of schemes for processor synchronization, resource allocation, and processor interconnection. These design decisions result in trade-offs between cost, performance, and programmability.

Processor Synchronization

The processors in the system must *synchronize* with one another in order to communicate. With *control-driven* synchronization, the processors transmit and accept values at points in time determined by external control signals. For example, with *lock-step* synchronization the processors are periodically synchronized by a central controller for the purpose of exchanging data. Between synchronization points each processor executes a small code segment based on the newly received data. With lock-step synchronization, a time-consuming computation for one portion of the mesh will cause most of the system to remain idle until this computation is completed.

In a system based on data-driven synchronization the processors independently execute their own instruction streams waiting only when data is needed from some other processor. A processor sends data to another as soon as it has been computed in a "packet" containing the data value and some identification of the data. Data-driven synchronization allows greater autonomy of processors and greater asynchrony in their operation. Small irregularities can be absorbed by nearby processors rather than cause global inefficiencies. Of course, data-driven synchronization does not guarantee that all processors will be fully utilized, but it provides an important step.

Data-driven synchronization also helps provide the flexibility of operation needed to support high level languages. By removing the global synchronization of processors we decrease the severity of the penalty paid by nonoptimal program implementations.

Processor Allocation

A large scale computer system contains a variety of resources for processing, storage, and communication. These resources must be allocated both in time and in space, with the optimal allocation depending on the configuration and speed of the system components as well as on the program itself. Thus, the subject of resource allocation is large and complex. For the purpose of this paper we will consider mainly the allocation of processing resources.

The spatial allocation of processors involves mapping the different activities to be performed onto the processors of the system. With *static* mapping, the spatial allocation is fixed before the program execution begins. PDE simulations, with their regular and well-defined structures suggest a variety of static mapping schemes such as one zone and/or one node per processor, or one row of zones per processor. As long as the size of the problem matches the size of the system, and the amount of computation per processor can be reasonably well equalized, this approach seems quite attractive.

With dynamic mapping the activities of the program are assigned to the processors as the execution proceeds. This approach would in principle maximize the utilization of the processors and allow for highly irregular and dynamically changing program structures. However, the difficulty of effectively mapping tasks onto processors and the overhead needed to perform this allocation may negate the potential benefits.

In addition to mapping the operations onto the processors, the operations of each processor must be ordered in time. This scheduling of tasks within each processor can occur either statically or dynamically. Static scheduling occurs in conventional processors where the order of instruction execution is fixed in advance. While this approach leads to simpler processor design, it is vulnerable to the same problems as lock-step synchronization when applied to multiple processor systems. Unless operations can be scheduled so that data arrives before the operation which needs them is initiated, a processor will sit idle even if it has other tasks to perform. Static scheduling

within a processor would require a detailed timing analysis of the program and would fail when computations exceed their expected time. A dynamic scheduling scheme, on the other hand, involves simply maintaining a task list and executing those tasks for which the data is present. The increased flexibility and performance of dynamic task scheduling within each processor will easily offset its overhead.

Processor Interconnection Schemes

A variety of interconnection schemes have been proposed for multiple processor systems [3, 7, 9]. Rather than discussing the details of each of these designs, we shall explore some of the properties of these interconnection schemes in the context of the problem at hand.

Some interconnection schemes such as trees, rings, and Cartesian grids favor local over long-distance communication, whereas others such as the routing networks of the MIT data flow machine [7] require the same communication delay between any pair of nodes. Those favoring local communication typically require fewer components (switches and wires) and allow faster communication in the local case but are slower in the long-distance case.

As was seen in Figure 2, the SIMPLE program shows a great deal of locality between zone computations. Many computations depend only on data local to the zone while many others require data from only neighboring zones. Thus a potential does exist for exploiting the locality of communication. The mapping of operations onto processors, however, must match the locality in the program to the locality in the communication system. The degree to which this can be achieved depends on the activity mapping scheme and the type of interconnection network.

With static spatial mapping, one can easily imagine mapping adjacent zone computations onto adjacent processors. If the program size and structure does not match the system size and structure, however, complete locality cannot be maintained. For example, if the program has a 60 by 160 zone mesh, it cannot be mapped onto a 100 by 100 array of processors while maintaining the locality of the program and utilizing as many processors as possible. If resources are allocated dynamically, on the other hand, the assignment function must map operations which are likely to communicate onto nearby processors. This would greatly complicate the resource allocation problem.

Finally, even SIMPLE requires some global communication for finding a minimum over all zones and for distributing scalar values. One may also want to restrict the number of redundant copies of data or code to save storage, thereby increasing the long-distance communication requirements. Thus, the delay incurred by long-distance communication cannot be too great, although no quantitative requirements have been derived yet.

In summary, the structure SIMPLE code at first glance suggests a simple interconnected array of processors. Such a scheme would minimize the cost and naturally reflect most of the data dependencies of the algorithm. After further study, however, one realizes that global communication would probably take too long with such a scheme, and the configuration would not tolerate program structures and processor assignment schemes which do not match this connectivity. Nonetheless, a fully uniform interconnection scheme does not seem to be required, nor could its inability to take advantage of locality be tolerated. Some elaboration on an interconnected array of processors seems the most cost effective.

Conclusion

Research in data flow has been inspired largely by theoretical models of computation and languages. Hence, programming languages have been studied thoroughly, and greater consensus has been reached on their design. Exercises in programming scientific programs such as SIMPLE in high level data flow languages have proved quite promising in terms of both ease and the amount of concurrency which is shown. It has become clear that programming languages for highly concurrent systems must break away from the sequential memory update model of the Von Neumann computer and instead allow programs to be expressed in a maximally concurrent, functional form.

Research in architecture to support the data flow model, on the other hand, has not coalesced into a well-defined body knowledge. Most efforts have been directed at specific architectures with particular biases in terms of generality, performance, and cost. In studying the range of possible architectures for partial differential

equation simulation, it has become apparent that data flow concepts can and indeed *should* be applied at a variety of different levels. At the lowest levels, the architecture would be specialized toward the types of problems to be solved in terms of configuration, processor allocation, and interconnection but would employ data-driven control and a dynamic scheduling of activities within processors. These classes of machines would still require a certain amount of effort in mapping a program onto a machine but would at least allow a much more abstract view than do existing high performance machines. At higher levels of sophistication the architecture would support a very abstract data flow mode and dynamically handle all problems of resource allocation. These machines would allow more general classes of programs and would be less affected by irregular program structures and less than optimal code. Which type of machine should be built depends largely on the nature of the problems to be solved, the sophistication of the user community, and the acceptable cost of a machine. For PDE simulations, with their high computational requirements and statically-defined, regular structures, a specialized machine with static activity mapping and limited processor interconnections may indeed prove the best choice.

Acknowledgements

Our work has been aided greatly by the help given by John Myers as a consultant to the MIT Laboratory for Computer Science. We are also grateful to Chris Hendrickson, Tim Rudy, and John Woodruff of Lawrence Livermore Laboratory for first developing the SIMPLE program and then explaining many of the fine points of PDE simulation.

References

[1] Ackerman, W., "Data Flow Languages," *Proceedings of the 1979 National Computer Conference*, AFIPS (1979).

[2] Ackerman, W., and J. Dennis, "VAL -- A Value-Oriented Algorithmic Language: Preliminary Reference Manual," Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass. (1979).

[3] Arvind, K. P. Gostelow, and W. Plouffe, *An Asynchronous Programming Language and Computing Machine*, University of California Irvine Technical Report TR-114a (December, 1978).

[4] Crowley, W. P., C. P. Hendrickson, and T. E. Rudy, *The SIMPLE Code*, Internal Report UCID-17715, Lawrence Livermore Laboratories, Livermore, Ca. (Feb., 1978).

[5] Davis, A. L., "A Loosely-Coupled Applicative Multi-Processing System," *Proceedings of the 1979 National Computer Conference*, AFIPS (June, 1979).

[6] Dennis, J. B., "First Version of a Data Flow Procedure Language," *Programming Symposium: Proceedings, Colloque sur la Programmation*, (B. Robinet, Ed.), *Lecture Notes in Computer Science 19* (1974), 362-376.

[7] Dennis, J. B., and D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," *The Second Annual Symposium on Computer Architecture: Conference Proceedings*, (January, 1975), 126-132.

[8] Gostelow, K. P., and R. E. Thomas, *Performance of a Data-Flow Computer*, University of California Irvine Technical Report TR-127 (April, 1979).

[9] Gritton, E. C., *et al*, *Feasibility of a Special Purpose Computer to Solve the Navier-Stokes Equations*, Rand Technical Report R-2183-RC, Rand Corporation, Santa Monica, Ca. (1977).

[10] Keller, R., S. Patil, and G. Lindstrom, "An Architecture for a Loosely-Coupled Parallel Processor," *Proceedings of the 1979 National Computer Conference*, AFIPS (1979).

[11] Richtmyer, and Morton, *Difference Methods for Initial Value Problems*, Wiley Interscience, New York (1967).

Figure 1. Block Diagram of SIMPLE

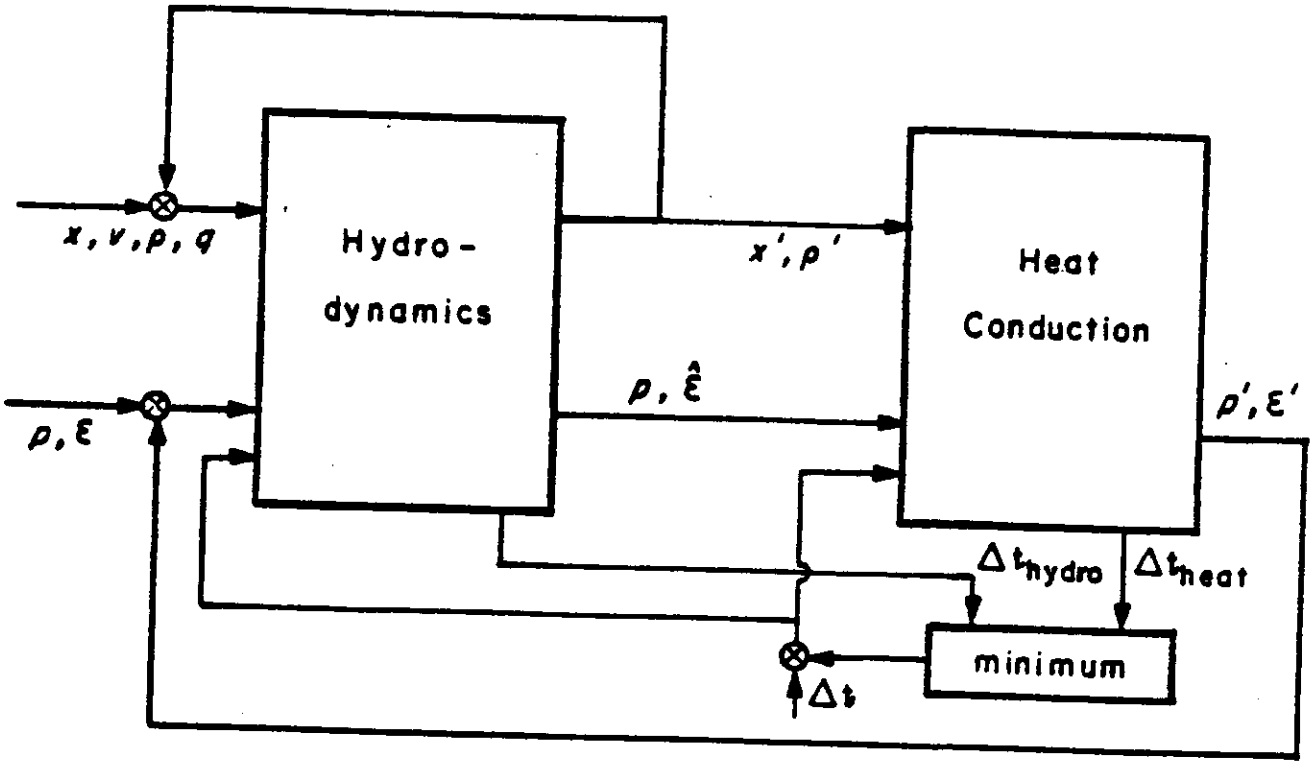
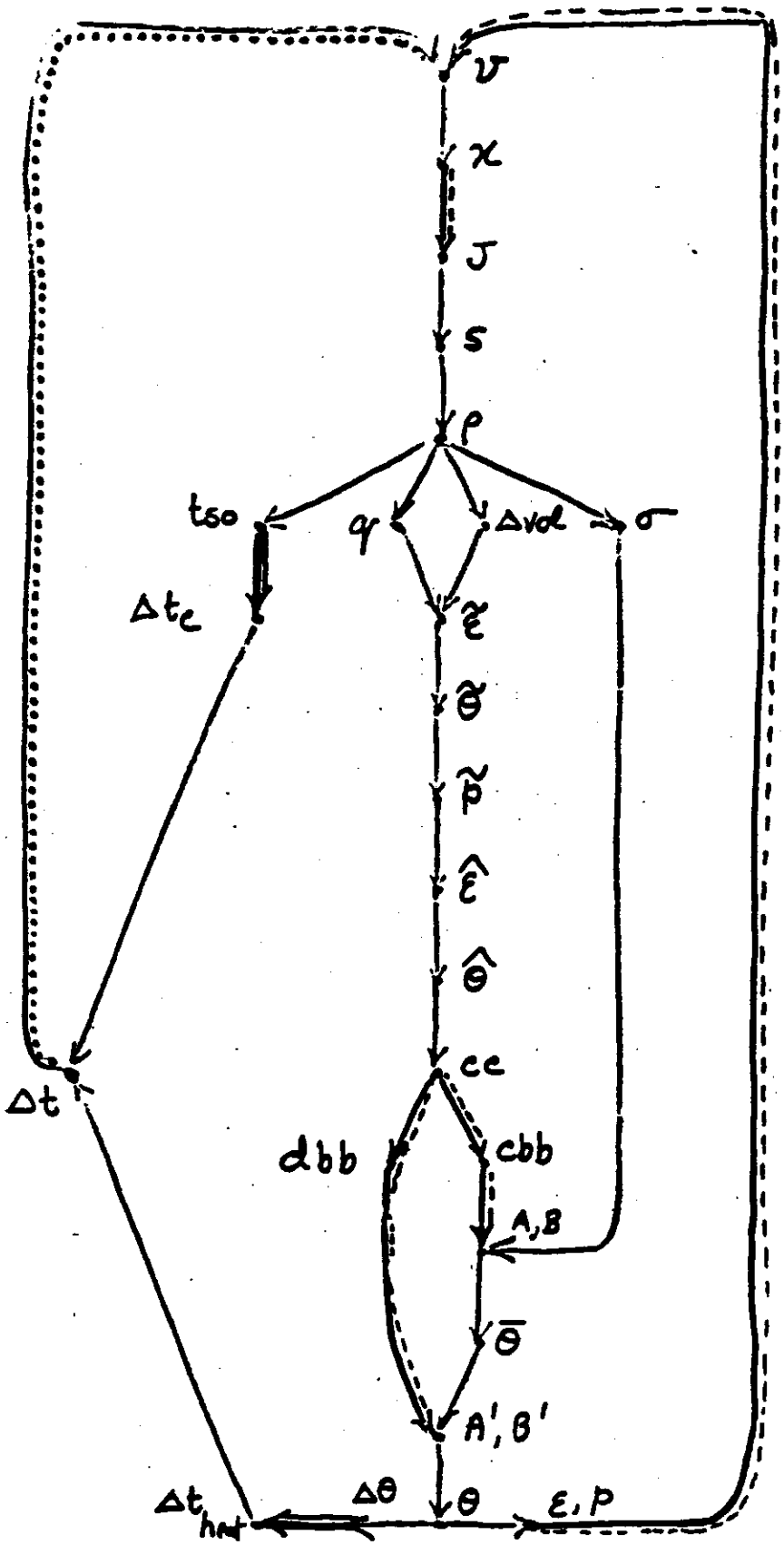


Figure 2. Data Dependencies in SIMPLE



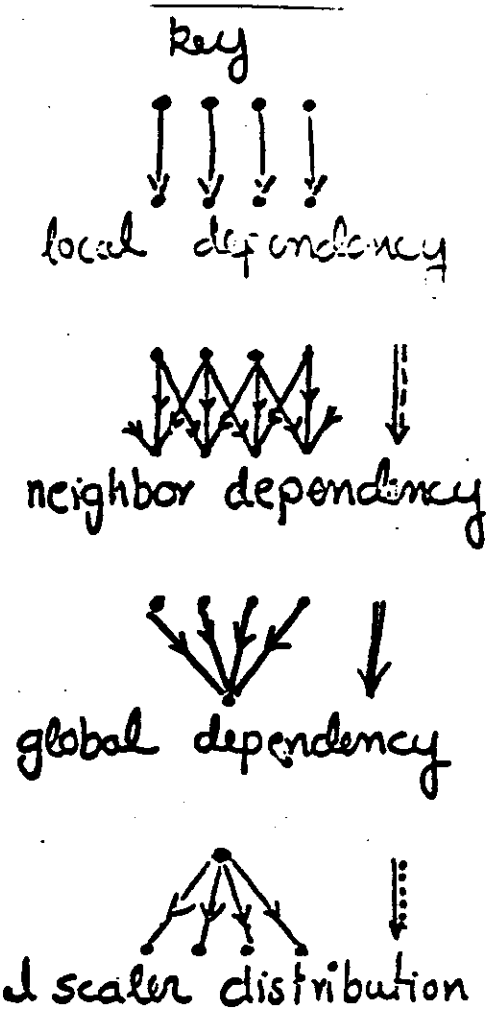
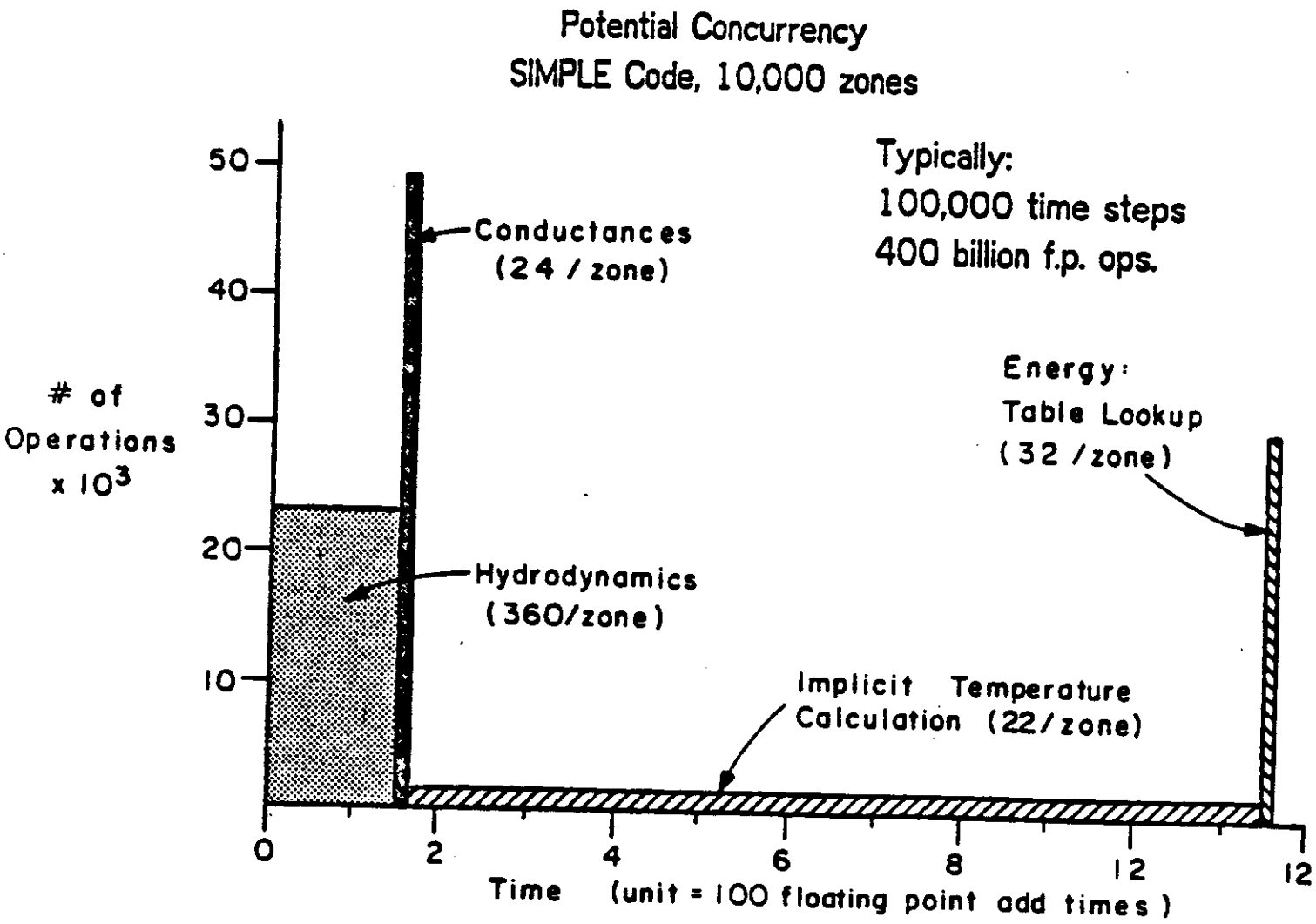


Figure 3.



Assumptions:

kmx, *lmx* = 100.

All floating point operations take one time unit.