FINAL REPORT
FOR
GRAPHICS APPLICATIONS UTILIZING PARALLEL PROCESSING

July 23, 1990

John R. Rice
Department of Computer Science
Purdue University
West Lafayette, IN 47907
FINAL REPORT
FOR
GRAPHICS APPLICATIONS UTILIZING
PARALLEL PROCESSING

Principal Investigator: Dr. John R. Rice
Department of Computer Science
Purdue University
West Lafayette, IN 47907

Technical Officer: Mr. Donald L. Lansing
NASA Langley Research Center
M/S 125A
Hampton, VA 23665-5225

Date: July 23, 1990

Grant Period: February 1, 1989 - June 30, 1990

Grant Number: NAG-1-789
This paper is the result of research conducted to develop a parallel graphic application algorithm to depict the numerical solution of the one-dimensional wave equation, the vibrating string. The research was conducted on a Flexible Flex/32 multiprocessor and a Sequent Balance 21000 multiprocessor. The wave equation is implemented using the finite difference method. The synchronization issues that arose from the parallel implementation and the strategies used to alleviate the effects of the synchronization overhead are discussed.

Introduction

The objective of this research is to develop a methodology for the implementation of a parallel graphic application algorithms for multiprocessor computers. The application algorithm used in this research is the one-dimensional wave equation, the vibrating string. The immediate goal of the research is to develop an algorithm that will solve and depict the numerical solution of the one-dimensional wave equation. The programming language used to implement the parallel graphic application algorithm is the Force programming language. The graphic routines implemented to depict the numerical solution of the one-dimensional wave equation are designed for use with the 4107 Tektronix Graphic Terminal.

The depiction of the numerical solution of the one-dimensional wave equation was chosen as the focus of this research because this equation is the foundation of all wave motion. The major research emphasis is placed on determining an approach for the depiction of the numerical solution. This approach involves determining the complications and benefits that are derived when the one-dimensional wave equation is implemented, solved, and depicted using a multiprocessor computer.

Portions of this research was conducted in two different multiprocessor environments. In one multiprocessing environment, multiple processors are dedicated to the execution of parallel programs, one program at a time. In the other environment, no processors are dedicated, and several programs (sequential and parallel) can be executed concurrently.

Each programming environment had it's impact on the execution of the implemented sequential and parallel algorithms. However, the major concern in both environments deal with the synchronization of the processors in order to achieve the desired results.

Approach

After the application was chosen, the depiction of the one-dimensional wave equation, a sequential algorithm was chosen that solved this equation. The original sequential algorithm was implemented and modified to meet some of the requirements imposed by the immediate goal of this research. The modifications to the implemented sequential algorithm were implemented in cycles that consist of 1) implementation, 2) testing, and 3) restructuring of the modified sequential algorithm. The restructuring of the
sequential implementation involves the addition of variables, and the reordering and addition of blocks of code. At an unspecified point following a series of these cycles, the results are recorded.

The parallel implementation is developed by adding parallel constructs, one at a time, to the implemented sequential version of the vibrating string algorithm, working from the exterior to the interior of the sequential algorithm. The parallel constructs are implemented in a series of cycles consisting of 1) implementation of a parallel construct, 2) testing, and 3) restructuring the algorithm. Testing the parallel implementation involves executing the parallel program with different numbers of processors. It should be noted that an implemented parallel algorithm that works for 2 processors may or may not work for 4, 8, or 16 processors due to synchronization problems and the lack of data coherency [Dubois] through the use of shared variables. The restructuring of the parallel implementation involves the addition of private and shared variables, and the reordering and addition of blocks of code.

The approach taken to develop the parallel implementation is performed in a lock-step manner with the sequential implementation. As the desired results are achieved in the sequential implementation, a parallel implementation is developed to achieve the same results. Once the desired results are achieved for the numerical solution of the one-dimensional wave equation in the parallel implementation, the graphic routines are implemented in the sequential implementation. The above series of cycles are repeated for sequential implementation as well as for the parallel implementation, respectively, in order to achieve the desired graphical results.

Equations

The general one-dimensional wave equation [Slater][Burden] has the form

$$\frac{\partial^2 u}{\partial t^2} - \lambda \frac{\partial^2 u}{\partial x^2} = 0.$$  \hspace{1cm} (1)

The general equation produces a depiction of the vibrating string with no oscillating motion, a standing wave[Slater]. In order to produce the desired oscillating motion, an external force [Slater], $F_{ext}$, is added to obtain

$$\frac{\partial^2 u}{\partial t^2} - \lambda \frac{\partial^2 u}{\partial x^2} = F_{ext}.$$  \hspace{1cm} (2)

A damping force [Thomas][Slater], $e^{-bt}$, is used to create a damped motion for the vibrating string.
\[
\frac{\partial^2 u}{\partial t^2} - \lambda \frac{\partial^2 u}{\partial x^2} = \frac{1}{e^{bt}} F_{ext}.
\]

(3)

This is the modified one-dimensional wave equation that is implemented. The dampening force causes the vibrating string to return to its initial resting state after a specific number of iterations.

The other equations used in this research are used to calculate the speedup\([\text{Oleinick}]\)[\(\text{Quinn}\)] and efficiency\([\text{Quinn}\)] achieved by \(n_{\text{proc}}\) processors,

\[
speedup(n_{\text{proc}} \text{ processor(s)}) = \frac{\text{runtime(1 processor)}}{\text{runtime}(n_{\text{proc}} \text{ processor(s)})} \tag{4}
\]

\[
efficiency(n_{\text{proc}} \text{ processor(s)}) = \frac{\text{speedup}(n_{\text{proc}} \text{ processor(s)})}{n_{\text{proc}} \text{ processor(s)}} \tag{5}
\]

where \(n_{\text{proc}}\) represents the number of processors.

Programming environment

The programming language used to implement the sequential and parallel graphic application algorithms was the Force programming language [Jordan]. The Force programming language is an extended version of Fortran 77 with parallel constructs. The Force language is also portable. The sequential and parallel algorithms were initially developed on the Flexible Flex/32 multiprocessor. The Flex/32 was a multiprocessor that initially contained 20 processors. Two of the processors were dedicated as front end processors, and the remaining 18 processors were dedicated to the concurrent execution of parallel programs, one program at a time.

When the Flex/32 multiprocessor was no longer available, the implemented sequential and parallel algorithms were ported to a Sequent Balance 21000 multiprocessor with 16 processors. The portability of the Force programming language required minor changes to the sequential and parallel implementations. The Sequent multiprocessor is used as a multiuser multiprocessor[Sequent]. This multiprocessor [Oleinick][DuBois][Quinn] can execute several programs (sequential or parallel) concurrently and all processors are treated as equals. These programs can be system or user programs. The Flexible and Sequent multiprocessor have Unix based operating systems. The graphic routines used to depict the solution were developed for the Tektronix 4107 graphic terminals.

Parallel Constructs

The parallel constructs[Jordan] supported by the Force programming language and incorporated in the parallel implementation are barriers, critical sections, parallel
loops, and private and shared variables. Busy waits (spin locks) [DuBois][Dining] are also used to insure synchronization among the processors. This construct was designed as a very tight loop that allows the spinning processor to resume execution with a fast response time to a corresponding semaphore[Dining].

The barrier construct [Jordan][Dubois] is used to insure that only one processor executes the block of code that it contains. This construct requires that all of the processors used to execute a parallel program pass through this construct and the last processor to reach the beginning of the barrier executes the sequential code the construct encloses. This construct is used to read all user input, to output the timing results, and to insure the stability of the numerical solution[Slater][Burden]. The critical sections are used to provide mutual exclusion[Dining] for some calculations performed on some shared variables. This construct is mainly used to implement counting semaphores[DuBois] that are associated with row computations and the incrementation of loop control variables.

Two different versions of parallel loops are implemented, preschedule and selfschedule[Jordan]. These different forms of a general DO-CONTINUE loop allow the body of a loop to be executed in parallel. In the preschedule loop, the work distribution is determined before the loop is executed and this distribution is based on the number of processor used to execute the parallel program. Each processor is assigned a predetermined number of iterations to execute. The processors are synchronized after the loop is completed. The selfschedule loop allows each processor to request more work as it's respective work is complete. This is an effort to achieve better load balancing among the processors.

A private copy of some of the shared variables that remained constant throughout the execution of the parallel implementation of wave equation are also stored as private variables in an effort to alleviate some of the possible bus traffic due to memory contention [Stenstrom][Bhuyan]. An example of this duplication concerns the use of the array dimensions. In the worst case, 16 processors may attempt to access a particular shared variable at the same time. All of the arrays used were implemented as a shared variable. In the case of concurrent access to the same array element, data coherency is maintained by enclosing the computations associated with these variables in critical sections [DuBois][Bhuyan]. Another example is computations involved in incrementing a counting semaphore.

Graphic Routines

The graphical routines implemented to depict the numerical solution of the one-dimensional wave equation are centered around the use of routines that manipulate pixels. Pixel manipulations were chosen because pixel operations are a fast way to display and modify images on the screen. These operations also give a more realistic view of the vibrating string as oppose to the use of line segments which are part of the vector graphics. The viewport used in these implemented algorithms are for the for the full on screen viewport supported by the Tektronix 4107 Graphic Terminal. Each pixel that is used to depict the numerical solution of the one-dimensional wave equation
corresponds to a location in the pixel viewport. The pixels that depict the numerical solution are viewed as a string of points that represent the vibration string (see Figures 24-29). The other pixels in the viewport are treated as background.

The pixel routines are implemented in such a way that each vibration (movement) of the string is part of the computations for a complete pixel viewport. This image, or viewport, is computed and the graphic escape sequences that represents each viewport is stored in an array that is used to store M different viewports, one representing each row of the M X N solution to the wave equation.

The viewport image is stored in a two-dimensional array called LINE (see Appendices A, B, C; subroutine Runlength Write). The semaphore used to notify the output processor of the completion of computations for the viewport corresponding to iteration J is a semaphore called CODARY. Based on the following numerical sequence, 0,1,2,...,M-1, if the viewport corresponding to row J+1 (iteration J+1) is completed prior to the completion for of the viewport corresponding to row J, the output processor enters a busy wait. Once the Jth viewport has been computed, stored and depicted, the output processor is now free to increment its counter to J+1. If the corresponding J+1 semaphore has not been set to 1, then the output processor spins until the semaphore has been set.

Implementation

The wave equation is a hyperbolic partial differential equation [Burden] that has boundary and initial conditions. The initial sequential algorithm [Burden] computed the boundary conditions first, then computed the initial conditions, rows 0 and 1. The algorithm then perform the computations for rows 2,3,...,M-1. This approach was followed in order to solve the M X N system of equations.

In the parallel implementation, each processor is provided with a copy of the dimensions of a M X N matrix, variables MM and NN. The matrix is used to store the numerical solution of the one-dimensional wave equation. The M X N matrix corresponds to M equations and N unknowns. The elements stored in each row of this matrix corresponds to each iteration of the vibrating string (see Appendices B and C). Each processor is also provided with a private copy of the constant variables that are used in the graphic routines [Tektronix].

The one-dimensional wave equation was initially implemented as a sequential program. Parallel constructs supported by the Force programming language were incorporated to implement the first parallel version of the sequential algorithm. This initial parallel version was centered around the preschedule loops. Another version using self-schedule loops was developed later.

The flowchart for the implemented sequential algorithm is shown in Figure 1. The initialization of variables entails reading all user input and performing all initial calculations related to these input variables. The initialization calculations include the stability
computations. These computations are used to determine if the user's input will produce a stable numerical solution. If the results of the stability computations indicate that the numerical solution will be unstable, the number of time subdivisions is incremented by a constant integer. If necessary, the stability computations are recalculated until the stability requirements are satisfied. The program timer is started before the initialization process is started.

The flowchart for the implemented parallel algorithm, shown in Figure 2, executes the code pertaining to the initialization of variables by enclosing the above computations and input in a barrier construct. The program timer is started at the same point, but each individual processor also has a timer associated with the amount of parallel code it executes. The individual timers are started after the processor executing the initialization code enclosed in the barrier construct has completed its task.

After the computations for the boundary and initial conditions have been completed, a completion flag is set to signal the output processor that all computations for row 0 and 1 are completed (see location A in Figure 2). This completion flag is implemented in the form of a counting semaphore. Once the count reaches NPROC-1, the output processor proceeds by computing pixel information, starting with row 0. As the output processor completes the pixels computations for row j, it depicts the results of these computations. This process is repeated by the output processor, for row 1, 2, ..., until a rendezvous has occurred among the NPROC processors. This rendezvous is discussed below.

The sequential and parallel implementations of the body of the loops used to compute the boundary and initial conditions are similar to the loop used to compute the interior points for rows 2, 3, ..., M-1 (see Tables 1, 2, 3). These tables show the sequential, preschedule, and selfschedule implementations of loop 25, respectively. The application of the finite difference method to Equation (3) produces the equations used to compute the numerical solution that is stored in the array, W(I,J). The use of the finite difference method leads to a series of multistep computations for the variables, W(I,J+1), as shown in Tables 1, 2, and 3. The computation for W(I,J+1) depends on the results from the computations for W(I-1,J), W(I,J), W(I+1,J), and W(I,J-1). This dependency dictated the approach taken in the development of the parallel implementations of the sequential implementation shown in Table 1.

The initial approach taken in the parallel implementation to determine the numerical solution required a large amount of synchronization. Each processor was allowed to perform all computations for an individual row. Due to the above computational dependencies required, another approach was implemented that allowed the computations for row J to be performed by NPROC-1 processors. This approach eliminated the dependencies among the processors and is shown in Tables 2 and 3 for the preschedule and selfschedule versions. The processors are synchronized after the completion of the parallel loops. This approach required that NPROC-1 processors, NPROC is the number of processors, compute a section points for each row. The number of points computed by each processor is based on the computed value stored in the variable CELSIZ.
There is a three-dimensional array, \( \text{HOLDER} \), that is used to store information pertaining to each element in the array \( \text{W} \), the array containing the numerical solution for the one-dimensional wave equation. The information stored in \( \text{HOLDER} \) are the x-coordinate, the y-coordinate, the integer value of \( \text{W}(I,J) \), and the pixel number of \( \text{W}(I,J) \). The pixel number of \( \text{W}(I,J) \) is the location in the pixel viewport [Tektronix] representing the xy-coordinate (see Tables 1, 2, 3).

The preschedule version of statement 25 shows a modified version of a preschedule loop, \( \text{DO 30 - End presched DO} \), that uses NPROC-1 processor (Table 2). The variable \( \text{ME} \) is a private variable that is used to store the processor's id. The critical section, Critical XX, is used to implement a counting semaphore, \( \text{COUNT}(J) \). This counting semaphore is used to signal the output processor that rows 2, 3, ..., M-1 have been computed (Figure 2). This set of counting semaphores correspond to the setting of the completion flags at location B in Figure 2. Statement 31 and the statement that immediately follows in Tables 2 and 3 form the implementation of a busy wait (spin lock). This busy wait is used to prevent processors from performing unnecessary computations before the row variable, \( J \), is incremented.

The self-schedule version of statement 25, in Table 3, shows a modified version of a self-schedule loop that uses NPROC-1 processors. The critical sections, Critical XYZ30, are used to increment the loop control variable, \( I \), that represents the number of points calculated for each row \( J \).

When the variable RENDEZ is set to 1 (see Table 2 and 3), a rendezvous has occurred between the NPROC processors. The variable RENDEZ in the critical section, Critical XXX, is used to signal the completion of all computations pertaining to the interior points for rows 2, 3, ..., M-1. At this instance, all NPROC processors may be computing pixel information (see Figure 2, location B for NPROC-1 processor(s) and 1 processor). This is the only point in the execution of the implemented parallel algorithms that the NPROC processors may be executing the same segment of code.

The computation of pixel information uses the information stored in the three-dimensional array, \( \text{HOLDER} \). These computations include the computation of the color of each pixel, and the execution of the graphic routines that are used to depict the numerical solution. The number of colors supported by the graphic terminal used in this research is 16.

As the output processor is computing pixel information and depicting the results, it stores the index of the each row in the variable VOUS as it completes the corresponding row computations. Once the rendezvous has occurred, the output processor finishes it's present computations for some row \( J \) and ceases to compute pixel information (see Figure 2, locations C). There is a set of semaphores, \( \text{CODARY}(J) \), that correspond to the completion of the computations pertaining to pixel information for row \( J, J = \text{VOUS}+1, \text{VOUS}+2, ..., M-1 \) (see Appendices B and C, statement 88). At this point, the pixel computations for each row are performed by an individual processors since all of the row dependencies have been eliminated.
The sequential version's depiction of the numerical solution follows a flow of control that is similar to initial pixel computations and depictions performed by the output processor, Figure 1. After the computation of pixel information for row J, the solution is depicted. This process is continued for rows $J = 0, 1, ..., M-1$.

Results

The results displayed in this paper are obtained from the execution of the implemented sequential and parallel algorithms on the Sequent Balance 21000. In order to record the execution time on a multiuser multiprocessor, the best execution time is recorded out of a series of executions. In the case of a dedicated multiprocessor such as the Flexible Flex/32 multiprocessor, an average is taken of a series of execution times for a different number of processors, respectively. The results shown in Figures 4-23 represent the execution of 50 iterations of the 100 x 100 and 400 x 400 systems of equations. These figures are based on the execution times for 1, 2, 4, 8, and 16 processors. Using Equations 4 and 5, the speedups and efficiencies are computed. The work distributions for 2, 4, 8, and 16 processors are discussed.

The execution times are recorded for the sequential implementation and the two parallel implementations centered around the preschedule and self-schedule loops. The execution times in figures 4, 7, 10, and 13 for 1 processor correspond to the execution times for the sequential implementation.

The charts in figures 4-9 represent the results associated with the execution times required to solve the 100 x 100 system of equations. The charts in figures 10-23 represent the results associated with execution times required to solve the 400 x 400 system of equations. The charts that represent the speedup in figures 5, 8, 11, and 14 show the speedup achieved for their respective systems of equations. The unfilled portion of those figures represent the desired linear speedup which is the same as the number of processors used to solve the system of equations.

The best efficiencies were achieved in the use of 8 processors to solve the 100 x 100 and the 400 x 400 systems of equations. In all cases, the efficiency of solving the system with 16 processors was approximately the same or less efficient as using 4 processors. The efficiency achieved in solving the 400 x 400 system of equations show that using of 4 processors is almost as efficient as using 8 processors. The efficiency of using 2 processors to solve the 100 x 100 and the 400 x 400 systems of equations is less than 50% efficient. This shows that the implemented parallel algorithm is not well suited to the execution by 2 processors.

The charts in figures 16-23 show that the work distributions for 2, 4, 8, and 16 processors in solving the 400 x 400 system of equations. The 2 and 4 processor work distributions show the most even distributions of work. It should be noted that in terms of the 8 and 16 processor work distributions, the self-schedule implementation has basically the same distributions as the preschedule implementation. The biggest difference is that
the selfschedule loop iterations are assigned based on request, whereas the preschedule loop iterations are always determined before the loop is executed.

The images shown in figures 24-29 show the damping effect on the vibrating string for the 40th and 50th iterations. The damping force is initially applied to a 200 X 200 system of equations which is treated as the median between the 100 X 100 and 400 X 400 systems of equations. Figures 24 and 27 show the damping force applied to 100 X 100 system of equations. Figures 25 and 28 show the damping force applied to 200 X 200 system of equations. And, figures 26 and 29 show the damping force applied to the 400 X 400 system of equations.

Conclusion

The overall execution time required to solve the 100 X 100 system of equations using 2, 4, 8, and 16 processors was more efficient using the preschedule loops as compared to the self-schedule loops. This is not the case for the execution time required to solve the 400 X 400 system of equations. The synchronization overhead that is associated with the selfschedule loops is higher than overhead associated with the preschedule loops when the workload for the processors is small. However, as the workload for the processors increases, the selfscheduled implementation becomes more efficient. This increase in efficiency is due to the load balancing associated with the use of selfschedule loops.

The load balancing associated with the use of selfscheduled loops can be beneficial in the execution of parallel programs. Some of the problems associated with the use of multiprocessors, such as bus and memory contention, synchronization overhead, etc., can be offset through the use of the load balancing associated with selfschedule loops. In the case of preschedule loops, if any of the processors that have been assigned a large share of the work are delayed for any reason during program execution, these delays are reflected in the overall execution time. The selfschedule loops are an attempt to alleviate the effects of any of the above execution delays.

A major benefit of using a portable language such as the Force is that as one multiprocessor is no longer is available, another multiprocessor that is compatible to the environment required by the Force programming language can be used. However, this benefit can also be detrimental to the efficient execution of the implemented parallel algorithms if the type of multiprocessor architecture is not taken into consideration. When a new multiprocessor is needed to continue the development of parallel algorithms, it may be necessary to fine tune the system in order to achieve the most efficient execution of the implemented algorithms. Some multiprocessors may have processors dedicated to the execution of parallel programs such as the Flex multiprocessor. Other multiprocessors may be multi-user multiprocessors, such as the Sequent multiprocessor. Each type of multiprocessor has its advantages and disadvantages, but it is up to algorithm designer to make use of the fine tuning routines provided by the operating system in order to achieve maximum throughput for parallel implementations.
The major obstacle in designing an efficient parallel algorithm for any application is determining the best approach for work distribution coupled with minimal synchronization among processors. Normally, work is divided in conjunction with the execution of loops. When solving a system of equations and depicting the numerical solution, it may be necessary to devise several threads of concurrent execution within one program.

In order to develop the best possible parallel graphic application algorithm for any application, the approach should be to initially develop a sequential implementation that solves and depicts the numerical solution of the application. Followed by performing timing studies on different segments of the sequential implementation. The segments of the implemented algorithm that are the most time consuming are possible candidates for potential incorporation of parallel constructs.

Based on the nature of the application being solved, the way that the sequential implementation is partitioned can lead to the development of different threads of execution in the parallel implementation. An example is another approach to the depiction of the vibrating string. It is now apparent that the most time consuming portions of the implemented algorithms are associated with the computations related to the execution of the graphic routines. With this knowledge, the main emphasis is now placed incorporating parallelism in the execution of these routines. It should be noted that each processor that executes the graphic routines performs the computations an individual pixel viewport, which is equivalent to one iteration of the vibrating string.

Since the need for synchronization has been eliminated in the execution of the graphics routines, the majority of the processors should be assigned this task from the start of the parallel implementation. The synchronization associated with the execution of the boundary conditions can be removed and the task of computing these boundary conditions can be assigned to the output processor. This is one thread of execution. Another thread of execution can be associated with the computations for the interior points. This task can also be assigned to one processor which will eliminate some synchronization overhead. Once this processor has completed the task of computing the interior points, it can join the other processors that are performing the computations associated with the graphic routines. There will still be some synchronization overhead associated with this approach. However, the emphasis is placed on achieving higher throughput.

In order to achieve the individual threads of execution, some of the parallel constructs support by the parallel programming language may need to be modified. As in the case of the Force language, the language has parallel constructs that are designed for the parallel execution of loops and procedures. In order to achieve the different threads of execution, the programmer must make use of the processor id in order to achieve the desired results.

There are several factors that affect the performance of the implemented algorithms. One factor that had a major impact on the performance of the implemented
algorithms was the priorities given to each process. The execution priorities were always very low. These low execution priorities allowed the processes assigned to each processor to be swapped out when a process with a higher priority is encountered. This swapping process can impact the total execution of the parallel implementations if some form of synchronization is required during this swapping process. In the worst case, NPROC-1 processors are awaiting a response from a processor that has been put to sleep due to the swapping process. A higher priority number should have an impact on the required execution time.

Another factor affecting the performance is the use of the counting semaphores that are used to synchronize the NPROC-1 computation processors and the output processor. The time required for synchronization can be reduced to allow a faster depiction of the solution of one-dimensional wave equation with less synchronization overhead. However, it should be noted that if the synchronization at the end of the self-schedule loop is relaxed too much, some processors will perform no work.

In terms of the overall execution times recorded to obtain and depict the numerical solution of the one-dimensional wave equation, a very small portion of time is actually spend solving the system of equations. The majority of the execution time is spend performing the viewport computations. As the images become more complex than a vibrating string, more synchronization may be required which will have some effect on the performance of the implemented parallel algorithms used to depict the numerical solution of different types of equations.

There are system routines provided by the operating system of the Sequent multicomputer that facilitates the fine tuning of the operating system of for the execution of parallel programs. By fine tuning the system and eliminating some synchronization overhead, the efficiencies achieved for 2, 4, 8, and 16 processors should be improved.
REFERENCES


Table 1. Sequential version of loop 25

This loop is used to solve the one-dimensional wave equation. This loop calculates the values for rows 2 to M-1 (each processor has a private variable, MM).

25 CONTINUE
   J = JJJ
   T = J * K
   DO 26 II = I, MM-1
      X = II * H
      W(II, J+1) = 2.*(1.-LAMB2)*W(II, J)
      + LAMB2*(W(II+1, J)+W(II-1, J))
      + - W(II, J-1) + FORCE(external)
      + / e**(gamma*T)

   HOLDER(J+1, 0, II) = x-coordinate computation (based on X)
   HOLDER(J+1, 2, II) = int(W(II, J+1))
   HOLDER(J+1, 1, II) = y-coordinate computation (based on HOLDER(J+1, 2, II))
   HOLDER(J+1, 3, II) = pixel_number computation (based on HOLDER(J+1, 1, II) and HOLDER(J+1, 0, II))

26 CONTINUE
   JJJ = JJJ + 1

IF (JJJ.NE.NN) GO TO 25
Table 2. Preschedule version of loop 25

CELSIZ = INT((MM-1)/(NPROC-1)) + 1

Critical XYZ
COUNT(0) = COUNT(0) + 1
End critical

831  CONTINUE
     IF (COUNT(0).NE.(NPROC-1)) GO TO 831

This loop is used to solve the one-dimensional wave equation. This loop calculates the values for rows 2 to M-1 (each processor has a private variable, MM)

25  CONTINUE
J = JJJ
T = J * K
DO 30 I = (1) + ((CELSIZ))*(ME - 1), (MM-1),
   + ((CELSIZ))*(NPROC - 1)
   IST = (ME-1)*CELSIZ
   IEND = MIN(ME*CELSIZ, MM-1)
DO 26 II = IST, IEND
   X = II * H
   W(II,J+1) = 2.*(1.-LAMB2)*W(II,J)
   + LAMB2*(W(II+1,J)+W(II-1,J))
   + - W(II,J-1) + FORCE(external)
   / e**(gamma*T)
   HOLDER(J+1,0,II) = x-coordinate computation
                     (based on X)
   HOLDER(J+1,2,II) = int(W(II,J+1))
   HOLDER(J+1,1,II) = y-coordinate computation
                     (based on HOLDER(J+1,2,II))
   HOLDER(J+1,3,II) = pixel number computation
                     (based on HOLDER(J+1,1,II) and
                      HOLDER(J+1,0,II))
26  CONTINUE
30  End presched DO

Critical XX
IF ((COUNT(J)+1).EQ.(NPROC-1)) THEN
   JJJ = JJJ + 1
END IF
COUNT(J) = COUNT(J) + 1
End critical

31  CONTINUE
     IF (COUNT(J).NE.(NPROC-1)) GO TO 31
     IF (JJJ.NE.NN) GO TO 25
     COUNT(J+1) = COUNT(J)
     RENDEZ = 1
Table 3. Selfschedule version of loop 25

\[
\text{CELSIZ} = \text{INT}((\text{MM}-1)/(\text{NPROC}-1))+1
\]

Critical XYZ
\[
\text{COUNT}(0) = \text{COUNT}(0) + 1
\]
End critical

This loop is used to solve the one-dimensional wave equation. This loop calculates the values for rows 2 to M-1 (each processor has a private variable, MM)

25 CONTINUE
\[
\begin{align*}
J &= \text{JJJ} \\
T &= J \times K \\
\end{align*}
\]
Critical XYZ30
\[
\begin{align*}
\text{SELF30} &= \text{SELF30}+\text{CELSIZ} \\
I &= \text{SELF30} \\
\end{align*}
\]
End critical

30 CONTINUE
\[
\begin{align*}
\text{IST} &= (\text{ME}-1)\times\text{CELSIZ}+1 \\
\text{IEND} &= \text{MIN}(\text{ME} \times \text{CELSIZ}, \text{MM}-1) \\
\text{DO} 26 \text{II} &= \text{IST}, \text{IEND} \\
\text{X} &= \text{II} \times H \\
\text{W}(\text{II},\text{J}+1) &= 2 \times (1 - \text{LAMB2}) \times \text{W}(\text{II},\text{J}) \\
+ &+ \text{LAMB2} \times (\text{W}(\text{II}+1,\text{J}) + \text{W}(\text{II}-1,\text{J})) \\
+ &- \text{W}(\text{II},\text{J}+1) + \text{FORCE(external)} \\
&/ e^{\text{gamma}\times T}
\end{align*}
\]
\[
\begin{align*}
\text{HOLDER}(\text{J}+1,0,\text{II}) &= \text{x-coordinate computation (based on X)} \\
\text{HOLDER}(\text{J}+1,2,\text{II}) &= \text{int} (\text{W}(\text{II},\text{J}+1)) \\
\text{HOLDER}(\text{J}+1,1,\text{II}) &= \text{y-coordinate computation (based on HOLDER(J+1,2,II))} \\
\text{HOLDER}(\text{J}+1,3,\text{II}) &= \text{pixel_number computation (based on HOLDER(J+1,1,II) and HOLDER(J+1,0,II))}
\end{align*}
\]

26 CONTINUE
Critical XYZ30
\[
\begin{align*}
\text{SELF30} &= \text{SELF30}+\text{CELSIZ} \\
I &= \text{SELF30} \\
\end{align*}
\]
End critical

IF (I.LE. (MM-1)) GO TO 30

Critical XX
IF (\((\text{COUNT}(\text{J})+1).\text{EQ.}(\text{NPROC}-1)\) THEN
\[
\begin{align*}
\text{JJJ} &= \text{JJJ} + 1 \\
\text{CELSIZ} &= \text{INT}((\text{MM}-1)/(\text{NPROC}-1))+1 \\
\text{SELF30} &= -\text{CELSIZ}+1 \\
\text{IF (JJJ.EQ.NN) THEN} \\
\text{RENDEZ} &= 1 \\
\text{SELF90} &= \text{VOS} \\
\text{ENDIF} \\
\text{ENDIF} \\
\text{COUNT}(\text{J}) &= \text{COUNT}(\text{J}) + 1 \\
\end{align*}
\]
End critical

31 CONTINUE
IF (\((\text{COUNT}(\text{J}).\text{NE.}(\text{NPROC}-1))\) GO TO 31
IF (JJJ.NE.NN) GO TO 25

COUNT(J+1) = COUNT(J)
Figure 1. Flow chart for sequential one-dimensional wave equation.
Figure 2. Flow chart for parallel one-dimensional wave equation.
Figure 3. 4 point multi-step problem.
Figure 4. Preschedule Execution Time

Figure 5. Preschedule Speed-Up

Figure 6. Preschedule Efficiency
Figure 7. Selfschedule Execution Time

Figure 8. Selfschedule Speed-Up

Figure 9. Selfschedule Efficiency
Figure 16. Preschedule 2-Processor Work Distribution

Figure 17. Preschedule 4-Processor Work Distribution

Figure 18. Preschedule 8-Processor Work Distribution

Figure 19. Preschedule 16-Processor Work Distribution
Figure 24. Damping Effect on 100 x 100 System of Equations

Figure 25. Damping Effect on 200 x 200 System of Equations

Figure 26. Damping Effect on 400 x 400 System of Equations
Figure 27. Damping Effect on 100 x 100 System of Equations

Figure 28. Damping Effect on 200 x 200 System of Equations

Figure 29. Damping Effect on 400 x 400 System of Equations
This is the selfschedule version

Force WAVE of NPROC ident ME

String vibration program

Declarations

    Shared CHARACTER*15 LINE(0:51,0:800)
    Shared INTEGER INCNVAL, JJJ, M, N, RENDEZ
    Shared INTEGER TTBEG, TTEND, COUNT(0:800)
    Shared INTEGER HOLDER(0:400,0:4,0:401)
    Shared INTEGER LENGTH(0:51,0:800), VOUS, TIME(1:16)
    Shared INTEGER CODARY(0:800), SELF6, SELF20
    Shared INTEGER SELF30, SELF90, IT1, IT2, TIME1(1:16)
    Shared LOGICAL XX, XYZ, XYZ6, XYZ20
    Shared LOGICAL XYZ30, XYZ90
    Shared REAL L, TI, ALPHA
    Shared DOUBLE PRECISION LAMBDA, W(0:400,0:401)
Private CHARACTER*15 STRLINE
Common STRLINE, STRLEN
Private DOUBLE PRECISION LAMB2
Private INTEGER I, J, II, CELSIZ, SCREEN
Private INTEGER BITS, CODCOUN, XEND, YEND
Private INTEGER IST, IEND, MAXIM, MINIM, STRLEN
Private INTEGER MAXIXC, MULT, INDXPTR, INDXCOU
Private INTEGER COUN, MM, NN
Private INTEGER STOHOLD, IIHOLD
Private INTEGER CKHOLD, CK
Private INTEGER CKSLOPE, CKSLOPI, FLAG, FLAG22, FLAG33
Private INTEGER LBEG, LEND, COLOR
Private REAL H, X, K, T, SLOPE
Private REAL TEMP

End declarations

Barrier

Begin program timer

    TTBEG = timer()
    CALL PXBEGIN(1, I1, 4)
    CALL PXVIEW(0, 0, 639, 479)

Input of the length of the string.

    WRITE(6, *) 'Enter the length of the string: '
    READ *, L
    WRITE(6, *) L

Input of the time limitation.

    WRITE(6, *) 'Enter the time limit: '
    READ *, TI
    WRITE(6, *) TI

Input of the number of subdivisions for the string.

    WRITE(6, *) 'Enter the number of subdivisions for the string: '
    READ *, M
    WRITE(6, *) M

Input of the number of subdivisions for the time.

    WRITE(6, *) 'Enter the number of subdivisions for the time: '
    READ *, N
    WRITE(6, *) N
WRITE(6,*) 'Enter the number of time subdivisions: '
READ *, N
WRITE(6,*) N

Input of the value for alpha.

WRITE(6,*) 'Enter the value for alpha: '
READ *, ALPHA
WRITE(6,*) ALPHA

The following is used to insure convergence and stability of the numerical solution of the one-dimensional wave equation. The value of N, the number of time subdivisions, is incremented by 50 in a effort to insure convergence and stability.

LAMBDA = 0.
INCNVAL= N
SELF6= -1
SELF20= 0

CELSIZ= INT((M-I)/(NPROC-I))+I
SELF30= -CELSIZ+1

N= INCNVAL
H= L/M
K= TI/N
LAMBDA= K*ALPHA/H
INCNVAL= N + 50
IF (LAMBDA .GT. I.) GO TO 5

WRITE(6,*)
WRITE(6,*) 'The value of N is ',N
WRITE(6,*)

JJJ= 1
End barrier

Beginning of individual processor time.

IT1= timer()

MM= M
NN= N

The following private variables are initialized for use in the graphic routine RUNLENGTH WRITE.

XEND= 639
YEND= 479
BITS= 4
MULTR= 2**BITS
MAXIXC= INT(65535/MULTR)
SCREEN= (XEND+I)*(YEND+I)

This is the point where the NPROC-I computation processors are separated from the output processor.

IF (ME.NE.NPROC) THEN
H= L/MM
K= TI/NN
LAMB2= (K*ALPHA/H)**2
Limiting the output to 50 iterations.

NN = 50

This loop computes all of the boundary points for the vibrating string.

Modified implementation of a selfschedule DO-CONTINUE loop

Critical XYZ6
   SELF6 = SELF6+1
   J = SELF6
End critical

6 CONTINUE

   
   
   
   
   X = 0
   W(0,J) = SIN(3.1415927*0.)
   HOLDER(J,0,0) = int(X*100+10)
   HOLDER(J,2,0) = int(W(0,J))
   HOLDER(J,1,0) = HOLDER(J,2,0)+240
   HOLDER(J,3,0) = (YEND-HOLDER(J,1,0))*(XEND+1)
   
   X = MM*H
   W(MM,J) = SIN(3.1415927*L)
   HOLDER(J,0,MM) = int(X*100+10)
   HOLDER(J,2,MM) = int(W(MM,J))
   HOLDER(J,1,MM) = HOLDER(J,2,MM)+240
   HOLDER(J,3,MM) = (YEND-HOLDER(J,1,MM))*(XEND+1)
   
   The following two values are used in the pixel color computations.

   HOLDER(J,0,MM+1) = int((MM+1)*H*100+10)
   HOLDER(J,1,MM+1) = HOLDER(J,1,MM)

   The initialization of the array associated with the counting semaphores for the completion of computations for the interior points for rows 0,1,...,M-1.

   COUNT(J) = 0

   Critical XYZ6
   SELF6 = SELF6+1
   J = SELF6
End critical

   IF (J.LE.NN) GO TO 6

This loop computes the initial conditions, the interior points for row 0 and row 1.

Modified implementation of a selfschedule DO-CONTINUE loop

Critical XYZ20
   SELF20 = SELF20+1
   II = SELF20
End critical

20 CONTINUE

   X = II*H
Row j=0 computations

\[ W(II,0) = \sin(3.1415927 \cdot II \cdot H) \]
\[ \text{HOLDER}(0,0,II) = \text{int}(x*100+10) \]
\[ \text{HOLDER}(0,2,II) = \text{int}(W(II,0)) \]
\[ \text{HOLDER}(0,1,II) = \text{HOLDER}(0,2,II) + 240 \]
\[ \text{HOLDER}(0,3,II) = (YEND - \text{HOLDER}(0,1,II)) \cdot (XEND+1) + \text{HOLDER}(0,0,II) + 1 \]

Row j=0 computations

\[ W(II,1) = (1.-\text{LAMB2}) \cdot W(II,0) \]
\[ + \frac{\text{LAMB2}}{2} \cdot (\sin(3.1415927 \cdot (II+1) \cdot H) - \sin(3.1415927 \cdot (II-1) \cdot H)) + K \cdot 0 \]
\[ \text{HOLDER}(1,0,II) = \text{int}(x*100+10) \]
\[ \text{HOLDER}(1,2,II) = \text{int}(W(II,1)) \]
\[ \text{HOLDER}(1,1,II) = \text{HOLDER}(1,2,II) + 240 \]
\[ \text{HOLDER}(1,3,II) = (YEND - \text{HOLDER}(1,1,II)) \cdot (XEND+1) + \text{HOLDER}(1,0,II) + 1 \]

Critical XYZ20
\[ \text{SELF20} = \text{SELF20} + 1 \]
\[ \text{II} = \text{SELF20} \]

End critical

IF (II.LE.(MM-I)) GO TO 20

\[ \text{CELSIZ} = \text{INT}((MM-I)/(NPROC-I)) + 1 \]

Critical XYZ
\[ \text{COUNT}(0) = \text{COUNT}(0) + 1 \]

End critical

25 CONTINUE
\[ J = J J \]
\[ T = J \cdot K \]

Modified implementation of a selfschedule DO-CONTINUE loop

Critical XYZ30
\[ \text{SELF30} = \text{SELF30} + \text{CELSIZ} \]
\[ \text{I} = \text{SELF30} \]

End critical

30 CONTINUE
\[ \text{IST} = (ME-1) \cdot \text{CELSIZ} + 1 \]
\[ \text{IEND} = \text{MIN}(\text{ME} \cdot \text{CELSIZ}, \text{MM}-1) \]
\[ \text{DO 26 II= IST, IEND} \]
\[ \text{X} = II \cdot H \]
\[ W(II,J+1) = 2. \cdot (1.-\text{LAMB2}) \cdot W(II,J) \]
\[ + \frac{\text{LAMB2}}{2} \cdot (W(II+1,J) + W(II-1,J)) \]
\[ + - W(II,J-1) \cdot \cos(2. \cdot 3.1415927 \cdot T) \]
\[ + / 2.71828182845**(110*T) \]

\[ \text{HOLDER}(J+1,0,II) = \text{int}(X*100+10) \]
\[ \text{HOLDER}(J+1,2,II) = \text{int}(W(II,J+1)) \]
26 CONTINUE

Critical XYZ30
SELF30 = SELF30 + CELSZI
I = SELF30
End critical

IF (I.LE. (MM-1)) GO TO 30

Critical XX
IF ((COUNT(J)+1).EQ.(NPROC-1)) then
JJJ = JJJ + 1
CELSZI = INT((MM-1)/(NPROC-1))+1
SELF30 = -CELSZI+1

IF (JJJ.EQ.NN) THEN

The occurrence of the processor rendezvous.

RENDZ = 1
SELF90 = VOUS
END IF

END IF
COUNT(J) = COUNT(J) + 1
End critical

31 CONTINUE
IF (COUNT(J).NE.(NPROC-1)) GO TO 31

IF (JJJ.NE.NN) GO TO 25

COUNT(J+1) = COUNT(J)

DO 90 J = (VOUS+1) + ME - 1, (NN), NPROC - 1

Critical XYZ90
SELF90 = SELF90 + 1
J = SELF90
End critical

IT2 = timer()
TIME(ME) = IT2-IT1

Modified implementation of a selfschedule DO-CONTINUE loop

90 CONTINUE
MAXIM = -65535
MINIM = 65535
CONTINUE
IF (COUNT(J).NE.(NPROC-1)) GO TO 3303

Computations for pixel colors based on slope computations.

FLAG = 0
FLAG22 = 0
FLAG33 = 0
DO 335 I = 0, MM
   IF (HOLDER(J,3,I).GT.MAXIM) MAXIM = HOLDER(J,3,I)
   IF (HOLDER(J,3,I).LT.MINIM) MINIM = HOLDER(J,3,I)
   SLOPE = (HOLDER(J,1,I+1)-HOLDER(J,1,I))
   IF (0.0 .NE. (HOLDER(J,0,I+1)-HOLDER(J,0,I))) THEN
      SLOPE = SLOPE / (HOLDER(J,0,I+1)-HOLDER(J,0,I))
   ELSE
      SLOPE = 0.0
   END IF
   TEMP = ABS(SLOPE)
   IF ((0.0.LE.TEMP).AND.(TEMP.LT.0.167)) THEN
      COLOR = 12
   ELSE IF ((0.167.LE.TEMP).AND.(TEMP.LT.0.333)) THEN
      COLOR = 4
   ELSE IF ((0.333.LE.TEMP).AND.(TEMP.LT.0.5)) THEN
      COLOR = 11
   ELSE IF ((0.5.LE.TEMP).AND.(TEMP.LT.0.667)) THEN
      COLOR = 10
   ELSE IF ((0.667.LE.TEMP).AND.(TEMP.LT.0.833)) THEN
      COLOR = 3
   ELSE IF ((0.833.LE.TEMP).AND.(TEMP.LT.1.0)) THEN
      COLOR = 9
   ELSE IF ((1.0.LE.TEMP).AND.(TEMP.LT.1.167)) THEN
      COLOR = 7
   ELSE IF ((1.167.LE.TEMP).AND.(TEMP.LT.1.333)) THEN
      COLOR = 8
   ELSE IF ((1.333.LE.TEMP).AND.(TEMP.LT.1.5)) THEN
      COLOR = 2
   ELSE IF ((1.5.LE.TEMP).AND.(TEMP.LT.1.667)) THEN
      COLOR = 15
   ELSE IF ((1.667.LE.TEMP).AND.(TEMP.LT.1.833)) THEN
      COLOR = 6
   ELSE
      COLOR = 1
   END IF
   IF (SLOPE.GT.0.0) THEN
      CKSLOPE = 1
   ELSE IF (SLOPE.EQ.0.0) THEN
      CKSLOPE = 0
   ELSE
      CKSLOPE = -1
   END IF
   IF ((FLAG.EQ.0).AND.(SLOPE.NE.0.0)) THEN
      CKSLOPE1 = CKSLOPE
      FLAG = 1
   END IF
   IF ((CKSLOPE.EQ.CKSLOPE1).OR.(CKSLOPE.EQ.0)) THEN
      HOLDER(J,4,I+1) = COLOR
   END IF
   IF (FLAG33.EQ.0) THEN
      FLAG22 = 0
      FLAG33 = 1
      HOLDER(J,4,I) = 12
   END IF
ELSE
   IF (FLAG22.EQ.0) THEN
      FLAG22 = 1
      FLAG33 = 0
      GO TO 335
   END IF
This section of the program is the inline encoding of the graphics routine, RUNLENGTH WRITE. This subroutine loads color indices into the pixel viewport.

CODCOUN= 0
CKHOLD= 0
CK= 0
MINIMUM= MINIM-1
STOHOLD= MAXIXC

14 CONTINUE

IF (STOHOLD.LT.MINIMUM) THEN
  STRLINE(I:)= CHAR(27)
  STRLINE(2:)= CHAR(82)
  STRLINE(3:)= CHAR(76)
  STRLEN= 3
  CALL DECCON(1)
  CALL DECCON(MULTR*MAXIXC+0)
  CODCOUN= CODCOUN+1
  LENGTH(J,CODCOUN)= STRLEN
  LINE(J,CODCOUN)(1:LENGTH(J,CODCOUN))= STRLINE(1:STRLEN)
  STOHOLD= STOHOLD+MAXIXC
  GO TO 14
ELSE
  STRLINE(I:)= CHAR(27)
  STRLINE(2:)= CHAR(82)
  STRLINE(3:)= CHAR(76)
  STRLEN= 3
  MINIMUM= MINIMUM-(STOHOLD-MAXIXC)
  CALL DECCON(1)
  CALL DECCON(MULTR*MINIMUM+0)
  CODCOUN= CODCOUN+1
  LENGTH(J,CODCOUN)= STRLEN
  LINE(J,CODCOUN)(1:LENGTH(J,CODCOUN))= STRLINE(1:STRLEN)
  STOHOLD= STOHOLD+MINIMUM
  INDXCOUN= 0
END IF

DO 140 INDXPTR= MINIM, MAXIM
  DO 1100 II= 0,MM
    IF (HOLDER(J,3,II).EQ.INDXPTR) THEN
      CK=I
      IIHOLD= II
      GO TO 199
    END IF
  CONTINUE

199 CONTINUE

IF (CK.EQ.1) THEN
  IF (INDXCOUN.EQ.0) GO TO 1917
  STRLINE(I:)= CHAR(27)
CALL DECCON (1)
CALL DECCON (MULTR*INDXCOU+0)
CODCOUN = CODCOUN+1
LENGTH (J, CODCOUN) = STRLEN
LINE (J, CODCOUN) (1:LENGTH (J, CODCOUN)) = STRLINE (1:STRLEN)
CONTINUE
STRLINE (1:) = CHAR (27)
STRLINE (2:) = CHAR (82)
STRLINE (3:) = CHAR (76)
STRLEN = 3
CALL DECCON (1)
CALL DECCON (MULTR*1+HOLDER (J, 4, I I HOLD))
CODCOUN = CODCOUN+1
LENGTH (J, CODCOUN) = STRLEN
LINE (J, CODCOUN) (1:LENGTH (J, CODCOUN)) = STRLINE (1:STRLEN)
INDXCOU = 0
CK = 0
ELSE IF ((INDXCOU.EQ.MAXIXC)
+ .OR. (INDXPTR.EQ.SCREEN)) THEN
STRLINE (1:) = CHAR (27)
STRLINE (2:) = CHAR (82)
STRLINE (3:) = CHAR (76)
STRLEN = 3
CALL DECCON (1)
CALL DECCON (MULTR*INDXCOU+0)
CODCOUN = CODCOUN+1
LENGTH (J, CODCOUN) = STRLEN
LINE (J, CODCOUN) (1:LENGTH (J, CODCOUN)) = STRLINE (1:STRLEN)
INDXCOU = 1
ELSE
INDXCOU = INDXCOU+1
END IF
CONTINUE
MINIMUM = MAXIM+1
STOHOLD = STOHOLD+MINIMUM-MINIM
CONTINUE
IF (STOHOLD.LT.SCREEN) THEN
STRLINE (1:) = CHAR (27)
STRLINE (2:) = CHAR (82)
STRLINE (3:) = CHAR (76)
STRLEN = 3
CALL DECCON (1)
CALL DECCON (MULTR*MAXIXC+0)
CODCOUN = CODCOUN+1
LENGTH (J, CODCOUN) = STRLEN
LINE (J, CODCOUN) (1:LENGTH (J, CODCOUN)) = STRLINE (1:STRLEN)
STOHOLD = STOHOLD+MAXIXC
GO TO 1444
ELSE
The following is the code executed by the output processor.

ELSE IF (ME.EQ.NPROC) THEN
    J = 0

Limiting the output to 50 iterations.

NN = 50

Checking the rendezvous flag.

CONTINUE
IF (RENDEZ.EQ.1) GO TO 88

VOUS = J
MAXIM = -65535
MINIM = 65535

CONTINUE
IF (COUNT(J).NE.(NPROC-I)) GO TO 303

FLAG = 0
FLAG22 = 0
FLAG33 = 0
DO 35 I = 0, MM
    IF (HOLDER(J, 3, I).GT.MAXIM) MAXIM = HOLDER(J, 3, I)
    IF (HOLDER(J, 3, I).LT.MINIM) MINIM = HOLDER(J, 3, I)

    SLOPE = (HOLDER(J, 1, I+1) - HOLDER(J, 1, I))
    IF (0.0 .NE. (HOLDER(J, 0, I+1) - HOLDER(J, 0, I))) THEN
        SLOPE = SLOPE/(HOLDER(J, 0, I+1) - HOLDER(J, 0, I))
    ELSE
        SLOPE = 0.0
    END IF

    TEMP = ABS(SLOPE)

    IF ((0.0.LE.TEMP).AND.(TEMP.LT.0.167)) THEN
        COLOR = 12
    ELSE IF ((0.167.LE.TEMP).AND.(TEMP.LT.0.333)) THEN
        COLOR = 4
    END IF
ELSE IF ((0.333.LE.TEMP).AND.(TEMP.LT.0.5)) THEN
  COLOR = 11
ELSE IF ((0.5.LE.TEMP).AND.(TEMP.LT.0.667)) THEN
  COLOR = 10
ELSE IF ((0.667.LE.TEMP).AND.(TEMP.LT.0.833)) THEN
  COLOR =  3
ELSE IF ((0.833.LE.TEMP).AND.(TEMP.LT.1.0)) THEN
  COLOR =  9
ELSE IF ((1.0.LE.TEMP).AND.(TEMP.LT.1.167)) THEN
  COLOR =  7
ELSE IF ((1.167.LE.TEMP).AND.(TEMP.LT.1.333)) THEN
  COLOR =  8
ELSE IF ((1.333.LE.TEMP).AND.(TEMP.LT.1.5)) THEN
  COLOR =  2
ELSE IF ((1.5.LE.TEMP).AND.(TEMP.LT.1.667)) THEN
  COLOR = 15
ELSE IF ((1.667.LE.TEMP).AND.(TEMP.LT.1.833)) THEN
  COLOR =  6
ELSE
  COLOR =  1
END IF

IF (SLOPE.GT.0.0) THEN
  CKSLOPE = 1
ELSE IF (SLOPE.EQ.0.0) THEN
  CKSLOPE = 0
ELSE
  CKSLOPE = -1
END IF

IF ((FLAG.EQ.0).AND.(SLOPE.NE.0.0)) THEN
  CKSLOPE = CKSLOPE
  FLAG = 1
END IF

IF ((CKSLOPE.EQ.CKSLOPE).OR. (CKSLOPE.EQ.0)) THEN
  HOLDER(J, 4, I+I) = COLOR
  IF (FLAG33.EQ.0) THEN
    FLAG22 = 0
    FLAG33 = 1
    HOLDER(J, 4, I) = 12
  END IF
ELSE
  IF (FLAG22.EQ.0) THEN
    FLAG22 = 1
    FLAG33 = 0
    GO TO 35
  END IF
  HOLDER(J, 4, I) = COLOR
END IF

CONTINUE

This section of the program is the inline encoding of graphics routine, RUNLENGTH WRITE. This subroutine loads color indices into the pixel viewport.

CODCOUN = 0
CKHOLD = 0
CK = 0
MINIMUM = MINIM -1
STOHOLD = MAXIXC

CONTINUE

IF (STOHOLD.LT.MINIMUM) THEN
  STRLINE(1:) = CHAR(27)
CALL DECCON(1)
CALL DECCON(MULTR*MAXIXC+0)

CODCOUN= CODCOUN+1
LENGTH(J,CODCOUN)= STRLEN
LINE(J,CODCOUN) (1:LENGTH(J,CODCOUN))= STRLINE(1:STRLEN)
STOHOLD= STOHOLD+MAXIXC

GO TO 214
ELSE
STRLINE(1:)= CHAR(27)
STRLINE(2:)= CHAR(82)
STRLINE(3:)= CHAR(76)
STRLEN= 3

MINIMUM= MINIMUM-(STOHOLD-MAXIXC)

CALL DECCON(1)
CALL DECCON(MULTR*MINIMUM+0)

CODCOUN= CODCOUN+1
LENGTH(J,CODCOUN)= STRLEN
LINE(J,CODCOUN) (1:LENGTH(J,CODCOUN))= STRLINE(1:STRLEN)
STOHOLD= STOHOLD+MINIMUM

INDXCOUN= 0
END IF

DO 40 INDXPITR= MINIM,MAXIM
DO 100 II= 0,MM
   IF (HOLDER(J,3,II).EQ.INDXPITR) THEN
      CK=I
      IIHOLD= II
      GO TO 99
   END IF
CONTINUE
CONTINUE

IF (CK.EQ.1) THEN
   IF (INDXCOUN.EQ.0) GO TO 917
   STRLINE(1:)= CHAR(27)
   STRLINE(2:)= CHAR(82)
   STRLINE(3:)= CHAR(76)
   STRLEN= 3

   CALL DECCON(1)
   CALL DECCON(MULTR*INDXCOUN+0)

   CODCOUN= CODCOUN+1
   LENGTH(J,CODCOUN)= STRLEN
   LINE(J,CODCOUN) (1:LENGTH(J,CODCOUN))= STRLINE(1:STRLEN)

   CONTINUE

917

   STRLINE(1:)= CHAR(27)
   STRLINE(2:)= CHAR(82)
   STRLINE(3:)= CHAR(76)
   STRLEN= 3

   CALL DECCON(1)
   CALL DECCON(MULTR*1+HOLDER(J,4,IIHOLD))
CODCOUN = CODCOUN + 1
LENGTH (J, CODCOUN) = STRLEN
LINE (J, CODCOUN) (1:LENGTH (J, CODCOUN)) = STRLINE (1:STRLEN)

INDXCOU = 0
CK = 0
ELSE IF ((INDXCOU.EQ.MAXIXC) .OR. (INDXPTR.EQ.SCREEN)) THEN
STRLINE (1:) = CHAR (27)
STRLINE (2:) = CHAR (82)
STRLINE (3:) = CHAR (76)
STRLEN = 3

CALL DECCON (1)
CALL DECCON (MULTR*INDXCOU+0)

CODCOUN = CODCOUN + 1
LENGTH (J, CODCOUN) = STRLEN
LINE (J, CODCOUN) (1:LENGTH (J, CODCOUN)) = STRLINE (1:STRLEN)

INDXCOU = 1
ELSE
INDXCOU = INDXCOU + 1
END IF
CONTINUE

MINIMUM = MAXIM + 1
STOHOLD = STOHOLD + MINIMUM - MINIM

CONTINUE

444 IF (STOHOLD.LT.SCREEN) THEN
STRLINE (1:) = CHAR (27)
STRLINE (2:) = CHAR (82)
STRLINE (3:) = CHAR (76)
STRLEN = 3

CALL DECCON (1)
CALL DECCON (MULTR*MAXIXC+0)

CODCOUN = CODCOUN + 1
LENGTH (J, CODCOUN) = STRLEN
LINE (J, CODCOUN) (1:LENGTH (J, CODCOUN)) = STRLINE (1:STRLEN)
STOHOLD = STOHOLD + MAXIXC
GO TO 444
ELSE
STRLINE (1:) = CHAR (27)
STRLINE (2:) = CHAR (82)
STRLINE (3:) = CHAR (76)
STRLEN = 3

CALL DECCON (1)
CALL DECCON (MULTR*(SCREEN-(STOHOLD-MAXIXC)) +0)

CODCOUN = CODCOUN + 1
LENGTH (J, CODCOUN) = STRLEN
LINE (J, CODCOUN) (1:LENGTH (J, CODCOUN)) = STRLINE (1:STRLEN)

INDXCOU = 0
END IF

CODARY (J) = CODCOUN

88 CONTINUE
IF (CODARY (J).EQ.0) GO TO 88
CALL PXPOSIT(0, 479)

DO 3 CODCOUN = 1, CODARY(J)
    WRITE (6, *) LINE(J, CODCOUN) (1: LENGTH(J, CODCOUN))
  3 CONTINUE

J = J + 1
IF (J .NE. NN + 1) GO TO 33

END IF

IT2 = timer()
TIME1(ME) = IT2 - IT1

WRITE(6, *)
WRITE(6, *)
Barrier
TTEND = timer()

DO 3333 I = 1, NPROC
    WRITE (6, *) 'Processor ', I
    WRITE (6, *)
    WRITE (6, *) 'Section1 time solving the problem = ', TIME(I)
    WRITE (6, *) 'Section time = ', TIME1(I)
    WRITE (6, *)
  3333 CONTINUE

WRITE (6, *) 'The total time is ', (TTEND - TTBEG)
End barrier

Join
END

SUBROUTINE DECCON(X)

C This graphics subroutine converts integer parameter in host syntax.
C
COMMON DE, CON
CHARACTER *15 DE
INTEGER X, ABSNUM, DEC, CON
INTEGER BIN, HI1, HI2, LO1, HI1DEC, HI2DEC, LO1DEC
DIMENSION BIN(0:15), HI1(0:6), HI2(0:6), LO1(0:6)
DIMENSION DEC(0:15)

C Initialization of arrays and local variables.
C
DO 5 K = 0, 6
    HI1(K) = 0
    HI2(K) = 0
    LO1(K) = 0
  5 CONTINUE
DO 10 K = 0, 15
    BIN(K) = 0
    DEC(K) = 2**K
  10 CONTINUE
HI1DEC = 0
HI2DEC = 0
LO1DEC = 0

C Converts the INTEGER parameter to binary.
ABSNUM = IABS(X)
DO 15 I = 15,0,-1
   IF (ABSNUM .GE. DEC(I)) THEN
      ABSNUM = ABSNUM - DEC(I)
      BIN(I) = 1
   ELSE IF (ABSNUM .EQ. 0) THEN
      GOTO 20
   END IF
CONTINUE
15

Assigning bits.

20  HI1(6) = 1
    HI2(6) = 1
    LO1(6) = 0
    LO1(5) = 1

   DO 25 J = 0,5
      HI1(J) = BIN(J+10)
      HI2(J) = BIN(J+4)
      IF (J .LE. 3) THEN
         LO1(J) = BIN(J)
      ENDIF
   CONTINUE
25

   IF (X .GE. 0) THEN
      LO1(4) = 1
   ENDIF

Calculating the ASCII decimal equivalent (ADE) for array of bits.

   DO 30 K = 0,6
      IF (HI1(K) .NE. 0) THEN
         HI1DEC = HI1DEC + DEC(K)
      ENDIF
      IF (HI2(K) .NE. 0) THEN
         HI2DEC = HI2DEC + DEC(K)
      ENDIF
      IF (LO1(K) .NE. 0) THEN
         LO1DEC = LO1DEC + DEC(K)
      ENDIF
   CONTINUE
30

Transmitting the converted parameter to the terminal.

   CON= CON + 1
   DE(CON:)= CHAR(HI1DEC)
   CON= CON + 1
   DE(CON:)= CHAR(HI2DEC)
   CON= CON + 1
   DE(CON:)= CHAR(LO1DEC)
RETURN
END

SUBROUTINE XYCON(L,M)

This graphics subroutine converts xy-coordinates in host syntax.

COMMON PACK,NUM
CHARACTER*15 PACK
INTEGER NUM
INTEGER L,M,HIYDEC,EXTDEC,LOYDEC,HIXDEC
INTEGER LOXDEC, ABSNUM, DEC, XBIN, YBIN, EXTRA
INTEGER HIY, LOY, HIX, LOX
DIMENSION XBIN(0:II), YBIN(0:II), EXTRA(0:6)
DIMENSION HIY(0:6), LOY(0:6), HIX(0:6), LOX(0:6)
DIMENSION DEC(0:15)

C Initialization of arrays and local variables.

DO 5 K = 0,11
  YBIN(K) = 0
  XBIN(K) = 0
5 CONTINUE

DO 10 K = 0,6
  EXTRA(K) = 0
  HIX(K) = 0
  HIY(K) = 0
  LOY(K) = 0
  LOX(K) = 0
10 CONTINUE

DO 13 K = 0,15
  DEC(K) = 2**K
13 CONTINUE

HIYDEC = 0
EXTDEC = 0
LOYDEC = 0
HIXDEC = 0
LOXDEC = 0

C Converts the INTEGER parameters to binary.

DO 15 K=1,2
  DO 20 I= 11,0,-1
    IF (ABSNUM .GE. DEC(I)) THEN
      ABSNUM = ABSNUM - DEC(I)
      IF (K .EQ. 1) THEN
        XBIN(I) = 1
      ELSE
        YBIN(I) = 1
      ENDIF
    ELSE IF (ABSNUM .EQ. 0) THEN
      GOTO 25
   ENDIF
  ENDIF
  20 CONTINUE
25 ABSNUM = IABS(M)
15 CONTINUE

C Assigning bits.

HIY(6) = 0
HIY(5) = 1
EXTRA(6) = 1
EXTRA(5) = 1
EXTRA(4) = 0
EXTRA(3) = YBIN(1)
EXTRA(2) = YBIN(0)
EXTRA(1) = XBIN(1)
EXTRA(0) = XBIN(0)
LOY(6) = 1
LOY(5) = 1
HIX(6) = 0
HIX(5) = 1
LOX(6) = 1
LOX(5) = 0
DO 30 J = 0, 4
  HIY(J) = YBIN(J+7)
  LOY(J) = YBIN(J+2)
  HIX(J) = XBIN(J+7)
  LOX(J) = XBIN(J+2)
CONTINUE

Calculating the ASCII decimal equivalent (ADE) for array of bits.

DO 35 K = 0, 6
  IF (HIY(K) .NE. 0) THEN
    HIYDEC = HIYDEC + DEC(K)
  ENDIF
  IF (HIX(K) .NE. 0) THEN
    HIXDEC = HIXDEC + DEC(K)
  ENDIF
  IF (LOY(K) .NE. 0) THEN
    LOYDEC = LOYDEC + DEC(K)
  ENDIF
  IF (LOX(K) .NE. 0) THEN
    LOXDEC = LOXDEC + DEC(K)
  ENDIF
  IF (EXTRA(K) .NE. 0) THEN
    EXTDEC = EXTDEC + DEC(K)
  ENDIF
CONTINUE

Transmitting the converted parameter to the terminal.

NUM = NUM + 1
PACK(NUM:) = CHAR(HIYDEC)
NUM = NUM + 1
PACK(NUM:) = CHAR(EXTDEC)
NUM = NUM + 1
PACK(NUM:) = CHAR(LOYDEC)
NUM = NUM + 1
PACK(NUM:) = CHAR(HIXDEC)
NUM = NUM + 1
PACK(NUM:) = CHAR(LOXDEC)
RETURN
END

SUBROUTINE PXBEGIN(SURNUM, ALU, BPPIX)

This graphics subroutine sets up the terminal for subsequent pixel operations.

COMMON PX, BEG
CHARACTER *15 PX
INTEGER SURNUM, ALU, BPPIX, BEG

PX(1:) = CHAR(27)
PX(2:) = CHAR(82)
PX(3:) = CHAR(85)
BEG = 3
CALL DECCON(SURNUM)
CALL DECCON(ALU)
CALL DECCON(BPPIX)
WRITE(6, *) PX(1:BEG)
RETURN
SUBROUTINE PXPOSIT(XLOW, YLOW)

This graphics subroutine sets up the position
of the pixel beam in the pixel viewport.

COMMON PX, POSIT
CHARACTER *15 PX
INTEGER XLOW, YLOW, POSIT
PX(1:) = CHAR(27)
PX(2:) = CHAR(82)
PX(3:) = CHAR(72)
POSIT = 3
CALL XYCON(XLOW, YLOW)
WRITE(6,*) PX(1:POSIT)
RETURN
END

SUBROUTINE PXVIEW(XLOW, YLOW, XHIGH, YHIGH)

This graphics subroutine specifies the pixel
viewport's size and position in graphics
memory.

COMMON PX, VIEW
CHARACTER *15 PX
INTEGER XLOW, YLOW, XHIGH, YHIGH, VIEW
PX(1:) = CHAR(27)
PX(2:) = CHAR(82)
PX(3:) = CHAR(83)
VIEW = 3
CALL XYCON(XLOW, YLOW)
CALL XYCON(XHIGH, YHIGH)
WRITE(6,*) PX(1:VIEW)
RETURN
END
This is the preschedule version

Force WAVE of NPROC ident ME

String vibration program

Declarations

Shared CHARACTER*15 LINE(0:51,0:800)
Shared INTEGER INCNVAL, JJJ, M, N, RENDEZ
Shared INTEGER TTBEG, TTEND, COUNT(0:800)
Shared INTEGER HOLDER(0:400,0:4,0:401)
Shared INTEGER LENGTH(0:51,0:800), VOUS
Shared INTEGER CODARY(0:800), IT1, IT2, TIME1(1:16)
Shared LOGICAL XX, XYZ
Shared REAL TT, ALPHA, L
Shared DOUBLE PRECISION LAMBDA, W(0:400,0:401)
Private CHARACTER*15 STRLINE
Common STRLINE, STRLEN
Private DOUBLE PRECISION LAMBDA2
Private INTEGER I, J, JJ, II, CELSIZ, SCREEN
Private INTEGER BITS, CODCOUN, XEND, YEND
Private INTEGER IST, IEND, MAXIM, MINIM, STRLEN
Private INTEGER MAXIXC, MULTX, INDXPTR, INDXCOU
Private INTEGER MM, N, MINIMUM
Private INTEGER STOHOLD, IIHOLD
Private INTEGER CKHOLD, CK
Private INTEGER CKSLOPE, CKSLOP1, FLAG, FLAG22, FLAG33
Private INTEGER COLOR
Private REAL H, X, K, T, SLOPE
Private REAL TEMP

Barrier

Begin program timer

TTBEG= timer()
CALL PXBEGIN(I,II,4)
CALL PXVIEW(0,0,639,479)

Input of the length of the string.

WRITE(6,*) 'Enter the length of the string: '
READ *, L
WRITE(6,*) L

Input of the time limitation.

WRITE(6,*) 'Enter the time limit: '
READ *, TI
WRITE(6,*) TI

Input of the number of subdivisions for the string.

WRITE(6,*) 'Enter the number of subdivisions for the string: '
READ *, M
WRITE(6,*) M

Input of the number of subdivisions for the time.

...
WRITE(6,*) 'Enter the number of time subdivisions: '
READ *,N
WRITE(6,*) N

Input of the value for alpha.

WRITE(6,*) 'Enter the value for alpha: '
READ *,ALPHA
WRITE(6,*) ALPHA

The following is used to insure the convergence and stability of the numerical solution of the one-dimensional wave equation. The value of N, the number of time subdivisions, is incremented by 50 in an effort to insure convergence and stability.

LAMBDA = 0.
INCNVAL= N

5 N= INCNVAL
H= L/M
K= TI/N
LAMBDA= K*ALPHA/H
INCNVAL= N + 50
IF (LAMBDA .GT. 1.) GO TO 5

WRITE(6,*)
WRITE(6,*) 'The value of N is ',N
WRITE(6,*)

JJJ= 1
End barrier

Beginning of individual processor timer

IT1= timer()
MM= M
NN= N

The following private variables are initialized for use in the graphic routine RUNLENGTH WRITE.

XEND= 639
YEND= 479
BITS= 4
MULTR= 2**BITS
MAXIXC= INT(65535/MULTR)
SCREEN= (XEND+I)*(YEND+I)

This is the point where the NPROC-I computation processors are separated from the output processor.

IF (ME.NE.NPROC) THEN
H= L/MM
K= TI/NN
LAMB2= (K*ALPHA/H)**2

Limiting the output to 50 iterations

NN= 50
This loop computes all of the boundary points for the vibrating string.

Modified preschedule DO-CONTINUE loop

```
DO 6 J = (0) + ME - 1, (NN), NPROC - 1
  X = 0
  W(0, J) = SIN(3.1415927*I0)
  HOLDER(J, 0, 0) = INT(X*I0 + I0)
  HOLDER(J, 2, 0) = INT(W(0, J))
  HOLDER(J, 1, 0) = HOLDER(J, 2, 0) + 240
  HOLDER(J, 3, 0) = (YEND - HOLDER(J, I, 0)) * (XEND + 1)
  + HOLDER(J, 0, 0) + I
  X = MM*I0
  W(MM, J) = SIN(3.1415927*I0)
  HOLDER(J, 0, MM) = INT(X*I0 + I0)
  HOLDER(J, 2, MM) = INT(W(MM, J))
  HOLDER(J, 1, MM) = HOLDER(J, 2, MM) + 240
  HOLDER(J, 3, MM) = (YEND - HOLDER(J, I, MM)) * (XEND + 1)
  + HOLDER(J, 0, MM) + I
```

The following two values are used in the pixel color computations.

```
HOLDER(J, 0, MM+I) = INT((MM+I)*I0 + I0)
HOLDER(J, I, MM+I) = HOLDER(J, I, MM)
```

The initialization of the array associated with the counting semaphores for the completion of computations for the interior points for rows 0, 1, ..., M-1

```
COUNT(J) = 0
End presched DO
```

This loop computes the initial conditions, the interior points for row 0 and row 1.

Modified preschedule DO-CONTINUE loop

```
DO 20 II = (I) + ME - 1, (MM-1), NPROC - 1
  X = II*H
Row j=0 computations
  W(II, 0) = SIN(3.1415927*II)*H
  HOLDER(0, 0, II) = INT(X*I0 + I0)
  HOLDER(0, 2, II) = INT(W(II, 0))
  HOLDER(0, 1, II) = HOLDER(0, 2, II) + 240
  HOLDER(0, 3, II) = (YEND - HOLDER(0, I, II)) * (XEND + 1)
  + HOLDER(0, 0, II) + I
Row j=1 computations
  W(II, 1) = (1-LAMB2)*W(II, 0)
  + LAMB2/2.
  + *(SIN(3.1415927*(II+1)*H)
  + SIN(3.1415927*(II-1)*H))
  + K*I0
  HOLDER(1, 0, II) = INT(X*I0 + I0)
```
HOLDER(1,2,II) = INT(W(II,1))
HOLDER(1,1,II) = HOLDER(1,2,II) + 240
HOLDER(1,3,II) = (YEND - HOLDER(1,1,II)) * (XEND + 1) + HOLDER(1,0,II) + 1

End presched DO

CELSIZ = INT((MM-1)/(NPROC-1)) + 1

Critical XYZ
COUNT(0) = COUNT(0) + 1
End critical

831 CONTINUE
IF (COUNT(0).NE.(NPROC-1)) GO TO 831

25 CONTINUE
J = JJJ
T = J * K

Modified preschedule DO-CONTINUE loop

DO 30 I = (I) + ((CELSIZ) * (ME - i)), (MM-I),
+ ((CELSIZ) * (NPROC-1))
IST = (ME-1) * CELSZ + 1
IEND = MIN(ME*CELSIZ, MM-1)
DO 26 II = IST, IEND
X = II * H
W(II, J+1) = 2 * (1 - LAMB2) * W(II, J)
+ LAMB2 * (W(II+1, J) + W(II-1, J))
+ - W(II, J-1) + COS(2 * 3.1415927 * T)
+ / 2.71828182845 ** (10 * T)

HOLDER(J+1,0,II) = INT(X * 100 + 10)
HOLDER(J+1,2,II) = INT(W(II, J+1))
HOLDER(J+1,1,II) = HOLDER(J+1,2,II) + 240
HOLDER(J+1,3,II) = (YEND - HOLDER(J+1,1,II)) * (XEND + 1)
+ HOLDER(J+1,0,II) + 1

26 CONTINUE
30 End presched DO

Critical XX
IF ((COUNT(J)+1).EQ.(NPROC-1)) THEN
JJJ = JJJ + 1
END IF
COUNT(J) = COUNT(J) + 1
End critical

31 CONTINUE
IF (COUNT(J).NE.(NPROC-1)) GO TO 31
IF (JJJ.NE.NN) GO TO 25
COUNT(J+1) = COUNT(J)

The occurrence of the processor rendezvous.
RENDEZ = 1

Modified preschedule DO-CONTINUE loop.

DO 90 J = (VOS+1) + ME - 1, (NN), NPROC - 1

MAXIM = -65535
MINIM = 65535
CONTINUE
IF (COUNT(J).NE.(NP0C-1)) GO TO 3303

Computations for pixel colors based on slope computations.

FLAG = 0
FLAG22 = 0
FLAG33 = 0
DO 335 I = 0, MM
   IF (HOLDER(J, 3, I).GT.MAXIM) MAXIM = HOLDER(J, 3, I)
   IF (HOLDER(J, 3, I).LT.MINIM) MINIM = HOLDER(J, 3, I)
   SLOPE = (HOLDER(J, I+1, I+1) - HOLDER(J, I, I))
   IF (0.0 .NE. (HOLDER(J, 0, I+1) - HOLDER(J, 0, I))) THEN
      SLOPE = SLOPE/(HOLDER(J, 0, I+1) - HOLDER(J, 0, I))
   ELSE
      SLOPE = 0.0
   END IF
   TEMP = ABS(SLOPE)
   IF ((0.0.LE.TEMP).AND.(TEMP.LT.0.167)) THEN
      COLOR = 12
   ELSE IF ((0.167.LE.TEMP).AND.(TEMP.LT.0.333)) THEN
      COLOR = 4
   ELSE IF ((0.333.LE.TEMP).AND.(TEMP.LT.0.5)) THEN
      COLOR = 11
   ELSE IF ((0.5.LE.TEMP).AND.(TEMP.LT.0.667)) THEN
      COLOR = 10
   ELSE IF ((0.667.LE.TEMP).AND.(TEMP.LT.0.833)) THEN
      COLOR = 3
   ELSE IF ((0.833.LE.TEMP).AND.(TEMP.LT.1.0)) THEN
      COLOR = 9
   ELSE IF ((1.0.LE.TEMP).AND.(TEMP.LT.1.167)) THEN
      COLOR = 7
   ELSE IF ((1.167.LE.TEMP).AND.(TEMP.LT.1.333)) THEN
      COLOR = 8
   ELSE IF ((1.333.LE.TEMP).AND.(TEMP.LT.1.5)) THEN
      COLOR = 2
   ELSE IF ((1.5.LE.TEMP).AND.(TEMP.LT.1.667)) THEN
      COLOR = 15
   ELSE IF ((1.667.LE.TEMP).AND.(TEMP.LT.1.833)) THEN
      COLOR = 6
   ELSE
      COLOR = 1
   END IF
   IF (SLOPE.GT.0.0) THEN
      CKSLOPE = 1
   ELSE IF (SLOPE.EQ.0.0) THEN
      CKSLOPE = 0
   ELSE
      CKSLOPE = -1
   END IF
   IF (((FLAG.EQ.0).AND.(SLOPE.NE.0.0)) THEN
      CKSLOPE = CKSLOPE
      FLAG = 1
   END IF
   IF (((CKSLOPE.EQ.CKSLOPE).OR.(CKSLOPE.EQ.0)) THEN
      HOLDER(J, 4, I+1) = COLOR
   END IF
335    CONTINUE
This section of the program is the inline encoding of graphics routine, RUNLENGTH WRITE. This subroutine loads color indices into the pixel viewport.

CODCOUN= 0
CKHOLD= 0
CK= 0
MINIMUM= MINIM-1
STOHOLD= MAXIXC

CONTINUE

IF (STOHOLD.LT.MINIMUM) THEN
  STRLINE(1::) = CHAR(27)
  STRLINE(2::) = CHAR(82)
  STRLINE(3::) = CHAR(76)
  STRLEN= 3
  CALL DECCON(1)
  CALL DECCON(MULTR*MAXIXC+0)
  CODCOUN= CODCOUN+1
  LENGTH(J,CODCOUN) = STRLEN
  LINE(J,CODCOUN) (1::LENGTH(J,CODCOUN)) = STRLINE(1::STRLEN)
  STOHOLD= STOHOLD+MAXIXC
  GO TO 14
ELSE
  STRLINE(1::) = CHAR(27)
  STRLINE(2::) = CHAR(82)
  STRLINE(3::) = CHAR(76)
  STRLEN= 3
  MINIMUM= MINIMUM-(STOHOLD-MAXIXC)
  CALL DECCON(1)
  CALL DECCON(MULTR*MINIMUM+0)
  CODCOUN= CODCOUN+1
  LENGTH(J,CODCOUN) = STRLEN
  LINE(J,CODCOUN) (1::LENGTH(J,CODCOUN)) = STRLINE(1::STRLEN)
  STOHOLD= STOHOLD+MINIMUM
  INDXCOU= 0
END IF

DO 140 INDXPTR= MINIM,MAXIM
  DO 1100 II= 0,MM
    IF (HOLDER(J,3,II).EQ.INDXPTR) THEN
      CK=1
      IIHOLD= II
      GO TO 199
  END IF
  1100  CONTINUE
  199  CONTINUE
IF (CK.EQ.1) THEN
  IF (INDXCOU.EQ.0) GO TO 1917
  STRLINE(I:) = CHAR(27)
  STRLINE(2:) = CHAR(82)
  STRLINE(3:) = CHAR(76)
  STRLEN = 3
  CALL DECCON(1)
  CALL DECCON(MULTR*INDXCOU+0)
  CODCOUN = CODCOUN+1
  LENGTH(J,CODCOUN) = STRLEN
  LINE(J,CODCOUN) (1:LENGTH(J,CODCOUN)) = STRLINE(1:STRLEN)
CONTINUE
  STRLINE(I:) = CHAR(27)
  STRLINE(2:) = CHAR(82)
  STRLINE(3:) = CHAR(76)
  STRLEN = 3
  CALL DECCON(1)
  CALL DECCON(MULTR*1+HOLDER(J,4,IIHOLD))
  CODCOUN = CODCOUN+1
  LENGTH(J,CODCOUN) = STRLEN
  LINE(J,CODCOUN) (1:LENGTH(J,CODCOUN)) = STRLINE(1:STRLEN)
  INDXCOU = 0
  CK = 0
ELSE IF ((INDXCOU.EQ.MAXIXC) .OR. (INDXPTR.EQ.SCREEN)) THEN
  STRLINE(I:) = CHAR(27)
  STRLINE(2:) = CHAR(82)
  STRLINE(3:) = CHAR(76)
  STRLEN = 3
  CALL DECCON(1)
  CALL DECCON(MULTR*INDXCOU+0)
  CODCOUN = CODCOUN+1
  LENGTH(J,CODCOUN) = STRLEN
  LINE(J,CODCOUN) (1:LENGTH(J,CODCOUN)) = STRLINE(1:STRLEN)
  INDXCOU = 1
ELSE
  INDXCOU = INDXCOU+1
END IF
CONTINUE
MINIMUM = MAXIM+1
STOHOLD = STOHOLD+MINIMUM-MINIM
CONTINUE
IF (STOHOLD.LT.SCREEN) THEN
  STRLINE(I:) = CHAR(27)
  STRLINE(2:) = CHAR(82)
  STRLINE(3:) = CHAR(76)
  STRLEN = 3
  CALL DECCON(1)
  CALL DECCON(MULTR*MAXIXC+0)
  CODCOUN = CODCOUN+1
  LENGTH(J,CODCOUN) = STRLEN
  LINE(J,CODCOUN) (1:LENGTH(J,CODCOUN)) = STRLINE(1:STRLEN)
STOHOLD= STOHOLD+MAXIXC

GO TO 1444

ELSE

  STRLINE(1:)= CHAR(27)
  STRLINE(2:)= CHAR(82)
  STRLINE(3:)= CHAR(76)
  STRLEN= 3

  CALL DECCON(1)
  CALL DECCON(MULTR*(SCREEN- (STOHOLD-MAXIXC))+0)

  CODCOUN= CODCOUN+1
  LENGTH(J, CODCOUN)= STRLEN
  LINE(J, CODCOUN) (1:LENGTH(J, CODCOUN))= STRLINE(1:STRLEN)

  INDXCOU= 0
END IF

CODARY(J)= CODCOUN

90  End presched DO

C The following is the code executed by the output processor.

C ELSE IF (ME.EQ.NPROC) THEN
  J= 0

C Limiting the output to 50 iterations.

C NN= 50

C Checking for the rendezvous flag

33  CONTINUE
  IF (RENDEZ.EQ.1) GO TO 88

  VOUS= J
  MAXIM= -65535
  MINIM= 65535

  303 CONTINUE
  IF (COUNT(J).NE.(NPROC-I)) GO TO 303

  FLAG= 0
  FLAG22= 0
  FLAG33= 0

  DO 35 I= 0,MM
    IF (HOLDER(J,3,I).GT.MAXIM) MAXIM= HOLDER(J,3,I)
    IF (HOLDER(J,3,I).LT.MINIM) MINIM= HOLDER(J,3,I)

    SLOPE= (HOLDER(J,1,I+1)-HOLDER(J,1,I))
    IF (0.0 .NE. (HOLDER(J,0,I+1)-HOLDER(J,0,I))) THEN
      SLOPE= SLOPE/(HOLDER(J,0,I+1)-HOLDER(J,0,I))
    ELSE
      SLOPE = 0.0
    END IF

    TEMP= ABS(SLOPE)

    IF ((0.0.LE.TEMP).AND.(TEMP.LT.0.167)) THEN
      COLOR= 12
    ELSE IF ((0.167.LE.TEMP).AND.(TEMP.LT.0.333)) THEN
      COLOR= 4
ELSE IF ((0.333.LE.TEMP).AND.(TEMP.LT.0.5)) THEN
COLOR= 11
ELSE IF ((0.5.LE.TEMP).AND.(TEMP.LT.0.667)) THEN
COLOR= 10
ELSE IF ((0.667.LE.TEMP).AND.(TEMP.LT.0.833)) THEN
COLOR= 3
ELSE IF ((0.833.LE.TEMP).AND.(TEMP.LT.1.0)) THEN
COLOR= 9
ELSE IF ((1.0.LE.TEMP).AND.(TEMP.LT.1.167)) THEN
COLOR= 7
ELSE IF ((1.167.LE.TEMP).AND.(TEMP.LT.1.333)) THEN
COLOR= 8
ELSE IF ((1.333.LE.TEMP).AND.(TEMP.LT.1.5)) THEN
COLOR= 2
ELSE IF ((1.5.LE.TEMP).AND.(TEMP.LT.1.667)) THEN
COLOR= 15
ELSE IF ((1.667.LE.TEMP).AND.(TEMP.LT.1.833)) THEN
COLOR= 6
ELSE
COLOR= 1
END IF

IF (SLOPE.GT.0.0) THEN
CKSLOPE = 1
ELSE IF (SLOPE.EQ.0.0) THEN
CKSLOPE = 0
ELSE
CKSLOPE = -1
END IF

IF ((FLAG.EQ.0).AND.(SLOPE.NE.0.0)) THEN
CKSLOP1 = CKSLOPE
FLAG= 1
END IF

IF ((CKSLOPE.EQ.CKSLOP1).OR.(CKSLOPE.EQ.0)) THEN
HOLDER(J, 4, I+1)= COLOR
IF (FLAG33.EQ.0) THEN
FLAG22= 0
FLAG33= 1
HOLDER(J, 4, I)= 12
END IF
ELSE
IF (FLAG22.EQ.0) THEN
FLAG22= 1
FLAG33 = 0
GO TO 35
END IF
HOLDER(J, 4, I)= COLOR
END IF

CONTINUE

This section of the program is the inline encoding of
graphic routine, RUNLENGTH WRITE. This subroutine
loads color indices into the pixel viewport.

CODCOUN= 0
CKHOLD= 0
CK= 0
MINIMUM= MINIM-1
STOHOLD= MAXIXC

CONTINUE
IF (STOHOLD.LT.MINIMUM) THEN
STRLINE(1:)= CHAR(27)
CALL DECCON(1)
CALL DECCON(MULTR*MAXIXC+0)

CODCOUN = CODCOUN+1
LENGTH(J,CODCOUN) = STRLEN
LINE(J,CODCOUN)(1:LENGTH(J,CODCOUN)) = STRLINE(1:STRLEN)
STOHOHD = STOHOHD+MAXIXC

GO TO 214

ELSE
STRLINE(1:) = CHAR(27)
STRLINE(2:) = CHAR(82)
STRLINE(3:) = CHAR(76)
STRLEN = 3

MINIMUM = MINIMUM-(STOHOHD-MAXIXC)

CALL DECCON(1)
CALL DECCON(MULTR*MINIMUM+0)

CODCOUN = CODCOUN+1
LENGTH(J,CODCOUN) = STRLEN
LINE(J,CODCOUN)(1:LENGTH(J,CODCOUN)) = STRLINE(1:STRLEN)
STOHOHD = STOHOHD+MINIMUM

INDXCOUN = 0
END IF

DO 40 INDXPTR = MINIM,MAXIM
  DO 100 II = 0,MM
    IF (HOLDER(J,3,II).EQ.INDXPTR) THEN
      CK = I
      IITHHD = II
      GO TO 99
    END IF
  CONTINUE
  CONTINUE
  IF (CK.EQ.1) THEN
    IF (INDXCOUN.EQ.0) GO TO 917
  STRLINE(1:) = CHAR(27)
  STRLINE(2:) = CHAR(82)
  STRLINE(3:) = CHAR(76)
  STRLEN = 3

  CALL DECCON(1)
  CALL DECCON(MULTR*INDXCOUN+0)

  CODCOUN = CODCOUN+1
  LENGTH(J,CODCOUN) = STRLEN
  LINE(J,CODCOUN)(1:LENGTH(J,CODCOUN)) = STRLINE(1:STRLEN)

  CONTINUE
  CONTINUE

  IF (CK.EQ.1) THEN
    IF (INDXCOUN.EQ.0) GO TO 917
  STRLINE(1:) = CHAR(27)
  STRLINE(2:) = CHAR(82)
  STRLINE(3:) = CHAR(76)
  STRLEN = 3

  CALL DECCON(1)
  CALL DECCON(MULTR*INDXCOUN+0)

  CODCOUN = CODCOUN+1
  LENGTH(J,CODCOUN) = STRLEN
  LINE(J,CODCOUN)(1:LENGTH(J,CODCOUN)) = STRLINE(1:STRLEN)
CALL DECCON(1)
CALL DECCON(MULTR*1+HOLDER(J,4,IIHOLD))

CODCOUN = CODCOUN+1
LENGTH(J,CODCOUN) = STRLEN
LINE(J,CODCOUN) (1:LENGTH(J,CODCOUN)) = STRLINE(1:STRLEN)

INDXCOU = 0
CK = 0
ELSE IF ((INDXCOU.EQ.MAXIXC).
. OR. (INDXPTR.EQ.SCREEN)) THEN
STRLINE(1:) = CHAR(27)
STRLINE(2:) = CHAR(82)
STRLINE(3:) = CHAR(76)
STRLLEN = 3
CALL DECCON(1)
CALL DECCON(MULTR*INDXCOU+0)
CODCOUN = CODCOUN+1
LENGTH(J,CODCOUN) = STRLEN
LINE(J,CODCOUN) (1:LENGTH(J,CODCOUN)) = STRLINE(1:STRLEN)

INDXCOU = 1
ELSE
INDXCOU = INDXCOU+1
END IF
CONTINUE
MINIMUM = MAXIM+1
STOHOLD = STOHOLD+MINIMUM-MINIM

CONTINUE
IF (STOHOLD.LT.SCREEN) THEN
STRLINE(1:) = CHAR(27)
STRLINE(2:) = CHAR(82)
STRLINE(3:) = CHAR(76)
STRLLEN = 3
CALL DECCON(1)
CALL DECCON(MULTR*MAXIXC+0)
CODCOUN = CODCOUN+1
LENGTH(J,CODCOUN) = STRLEN
LINE(J,CODCOUN) (1:LENGTH(J,CODCOUN)) = STRLINE(1:STRLEN)
STOHOLD = STOHOLD+MAXIXC
GO TO 444
ELSE
STRLINE(1:) = CHAR(27)
STRLINE(2:) = CHAR(82)
STRLINE(3:) = CHAR(76)
STRLLEN = 3
CALL DECCON(1)
CALL DECCON(MULTR*(SCREEN-(STOHOLD-MAXIXC))+0)
CODCOUN = CODCOUN+1
LENGTH(J,CODCOUN) = STRLEN
LINE(J,CODCOUN) (1:LENGTH(J,CODCOUN)) = STRLINE(1:STRLEN)
INDXCOU = 0
END IF
CODARY(J) = CODCOUN
CONTINUE
IF (CODARY(J).EQ.0) GO TO 88

Setting pixel starting position and depicting the solution.

CALL PXPOSIT(0,479)
DO 3 CODCOUN= 1,CODARY(J)
   WRITE(6,*) LINE(J,CODCOUN) (i : LENGTH(J,CODCOUN))
3 CONTINUE

J= J + 1
IF (J.NE.NN+1) GO TO 3

Stopping of individual processor timer

IT2= timer()
TIMEI(ME)= IT2-IT1
WRITE(6,*)
WRITE(6,*)

Output of timing results and stopping the program timer.

Barrier
TTEND= timer()
DO 3333 I = 1,NPROC
   WRITE(6,*) 'Processor ',I
   WRITE(6,*) 'Section time = ', TIME1(I)
3333 CONTINUE
WRITE(6,*)'The total time is ',(TTEND-TTBEG)
End barrier

Join
END

SUBROUTINE DECON(X)
This graphics subroutine converts integer parameter in host syntax.

COMMON DE,CON
CHARACTER *15 DE
INTEGER X,ABSNUM,DEC,CON
INTEGER BIN,HI1,HI2,LOI,HI1DEC,HI2DEC,LO1DEC
DIMENSION BIN(0:15),HI1(0:6),HI2(0:6),LO1(0:6)
DIMENSION DEC(0:15)

Initialization of arrays and local variables.

DO 5 K = 0,6
   HI1(K) = 0
   HI2(K) = 0
   LO1(K) = 0
CONTINUE
DO 10 K = 0,15
   BIN(K) = 0
   DEC(K) = 2**K
10 CONTINUE
HI1DEC = 0
HI2DEC = 0
LO1DEC = 0

Converts the INTEGER parameter to binary.
ABSNUM = IABS(X)
DO 15 I = 15,0,-1
   IF (ABSNUM .GE. DEC(I)) THEN
      ABSNUM = ABSNUM - DEC(I)
      BIN(I) = 1
   ELSE IF (ABSNUM .EQ. 0) THEN
      GOTO 20
   ENDIF
15 CONTINUE

Assigning bits.
HI1(6) = 1
HI2(6) = 1
LO1(6) = 0
LO1(5) = 1
DO 25 J = 0,5
   HI1(J) = BIN(J+10)
   HI2(J) = BIN(J+4)
   IF (J .LE. 3) THEN
      LO1(J) = BIN(J)
   ENDIF
25 CONTINUE

IF (X .GE. 0) THEN
   LO1(4) = 1
ENDIF

Calculating the ASCII decimal equivalent (ADE) for array of bits.
DO 30 K = 0,6
   IF (HI1(K) .NE. 0) THEN
      HI1DEC = HI1DEC + DEC(K)
   ENDIF
   IF (HI2(K) .NE. 0) THEN
      HI2DEC = HI2DEC + DEC(K)
   ENDIF
   IF (LO1(K) .NE. 0) THEN
      LO1DEC = LO1DEC + DEC(K)
   ENDIF
30 CONTINUE

Transmitting the converted parameter to the terminal.
CON = CON + 1
DE(CON:) = CHAR(HI1DEC)
CON = CON + 1
DE(CON:) = CHAR(HI2DEC)
CON = CON + 1
DE(CON:) = CHAR(LO1DEC)
RETURN
END
SUBROUTINE XYCON(L,M)

This graphics subroutine converts xy-coordinates in host syntax.

COMMON PACK,NUM
CHARACTER*15 PACK
INTEGER L,M,HIYDEC,EXTDEC,LOYDEC,HIXDEC
INTEGER LOXDEC,ABSNUM,DEC,XBIN,YBIN,EXTRA
INTEGER HIY,LOY,HIX,LOX
DIMENSION XBIN(0:II),YBIN(0:II),EXTRA(0:6)
DIMENSION HIY(0:6),LOY(0:6),HIX(0:6),LOX(0:6)
DIMENSION DEC(0:15)

Initialization of arrays and local variables.

DO 5 K = 0,II
   YBIN(K) = 0
   XBIN(K) = 0
5 CONTINUE

DO 10 K = 0,6
   EXTRA(K) = 0
   HIX(K) = 0
   HIY(K) = 0
   LOY(K) = 0
   LOX(K) = 0
10 CONTINUE

DO 13 K = 0,15
   DEC(K) = 2**K
13 CONTINUE

HIYDEC = 0
EXTDEC = 0
LOYDEC = 0
HIXDEC = 0
LOXDEC = 0

Converts the INTEGER parameters to binary.

ABSNUM = IABS(L)
DO 15 K=1,2
   DO 20 I= 11,0,-1
      IF (ABSNUM .GE. DEC(I)) THEN
         ABSNUM = ABSNUM - DEC(I)
         IF (K .EQ. 1) THEN
            XBIN(I) = 1
         ELSE
            YBIN(I) = 1
         ENDIF
      ELSE IF (ABSNUM .EQ. 0) THEN
         GOTO 25
      ENDIF
   20 CONTINUE
15 CONTINUE

Assigning bits.

HIY(6) = 0
HIY(5) = 1
Calculating the ASCII decimal equivalent (ADE) for array of bits.

DO 35 K = 0, 6
   IF (HIY(K) .NE. 0) THEN
      HIYDEC = HIYDEC + DEC(K)
   ENDIF
   IF (HIX(K) .NE. 0) THEN
      HIXDEC = HIXDEC + DEC(K)
   ENDIF
   IF (LOY(K) .NE. 0) THEN
      LOYDEC = LOYDEC + DEC(K)
   ENDIF
   IF (LOX(K) .NE. 0) THEN
      LOXDEC = LOXDEC + DEC(K)
   ENDIF
   IF (EXTRA(K) .NE. 0) THEN
      EXTDEC = EXTDEC + DEC(K)
   ENDIF
35   CONTINUE

Transmitting the converted parameter to the terminal.

NUM = NUM + 1
PACK(NUM:) = CHAR(HIYDEC)
NUM = NUM + 1
PACK(NUM:) = CHAR(EXTDEC)
NUM = NUM + 1
PACK(NUM:) = CHAR(LOYDEC)
NUM = NUM + 1
PACK(NUM:) = CHAR(HIXDEC)
NUM = NUM + 1
PACK(NUM:) = CHAR(LOXDEC)

RETURN
END

SUBROUTINE PXBEGIN(SURNUM, ALU, BPPIX)

This graphics subroutine sets up the terminal for subsequent pixel operations.

COMMON PX, BEG
CHARACTER *15 PX
INTEGER SURNUM, ALU, BPPIX, BEG

PX(1:) = CHAR(27)
PX(2:) = CHAR(82)
PX(3:) = CHAR(85)
BEG = 3
CALL DECCON(SURNUM)
CALL DECCON(ALU)
CALL DECCON(BPPIX)
WRITE(6,*) PX(1:BEG)
RETURN
END

SUBROUTINE PXPOSIT(XLOW, YLOW)
C
This graphics subroutine sets up the position
of the pixel beam in the pixel viewport.
C
COMMON PX, POSIT
CHARACTER *15 PX
INTEGER XLOW, YLOW, POSIT
PX(1:) = CHAR(27)
PX(2:) = CHAR(82)
PX(3:) = CHAR(72)
POSIT = 3
CALL XYCON(XLOW, YLOW)
WRITE(6,*) PX(1:POSIT)
RETURN
END

SUBROUTINE PXVIEW(XLOW, YLOW, XHIGH, YHIGH)
C
This graphics subroutine specifies the pixel
viewport's size and position in graphics
memory.
C
COMMON PX, VIEW
CHARACTER *15 PX
INTEGER XLOW, YLOW, XHIGH, YHIGH, VIEW
PX(1:) = CHAR(27)
PX(2:) = CHAR(82)
PX(3:) = CHAR(83)
VIEW = 3
CALL XYCON(XLOW, YLOW)
CALL XYCON(XHIGH, YHIGH)
WRITE(6,*) PX(1:VIEW)
RETURN
END
This is the sequential version

Force WAVE of NPROC ident ME

String vibration program

Declarations

Shared CHARACTER*15 LINE(0:51,0:800)
Shared INTEGER INCNVAL,JJJ,M,N
Shared INTEGER TTBEG,TTEND,COUNT(0:800)
Shared INTEGER HOLDER(0:400,0:4,0:401)
Shared INTEGER LENGTH(0:51,0:800),VOUS
Shared INTEGER CODARY(0:800),ITI,IT2,TIME1(1:16)
Shared REAL L,LI,ALPHA
Shared DOUBLE PRECISION LAMBDA,W(0:400,0:401)
Private CHARACTER*15 STRLINE
Common STRLINE,STRLEN
Private DOUBLE PRECISION LAMB2
Private INTEGER I,J,JJ,II,SCREEN
Private INTEGER BITS,CODCOUN,XEND,YEND
Private INTEGER MAXIM,MINIM,STRLEN
Private INTEGER MAXIXC,MULTR,INDXPTR,INDXCOU
Private INTEGER COUPN,MM,NN,PTRCOUN,MINIMUM
Private INTEGER STOHOLD,IIHOLD
Private INTEGER CHKOLD,CK
Private INTEGER CKSLOPE,CKSLOPI,FLAG, FLAG22,FLAG33
Private INTEGER LBEG, LEND,COLOR
Private REAL H,X,K,T,SLOPE
Private REAL TEMP

End declarations

Begin program timer

TTBEG = timer()

CALL PXBEGIN(I,II,4)
CALL PXVIEW(0, 0, 639, 479)

Input of the length of the string.

WRITE(6,*) 'Enter the length of the string: '
READ *,L
WRITE (6,*) L

Input of the time limitation.

WRITE(6,*) 'Enter the time limit: '
READ *,TI
WRITE (6,*) TI

Input of the number of subdivisions for the string.

WRITE(6,*) 'Enter the number of subdivisions for the string: '
READ *,M
WRITE (6,*) M

Input of the number of subdivisions for the time.

WRITE(6,*) 'Enter the number of time subdivisions: '
READ *,N
WRITE (6,*) N
Input of the value for alpha.

```
WRITE(6,*) 'Enter the value for alpha: ', ALPHA
WRITE(6,*) ALPHA
```

The following is used to insure the convergence and stability of the numerical solution of the one-dimensional wave equation. The value of N, the number of time subdivisions, is incremented by 50 in an effort to insure convergence and stability.

```
LAMBDA = 0.
INCNVAL = N
```

```
5 N = INCNVAL
H = L/M
K = TI/N
LAMBDA = K*ALPHA/H
INCNVAL = N + 50
IF (LAMBDA .GT. 1.) GO TO 5
WRITE(6,*)
WRITE(6,*) 'The value of N is ', N
WRITE(6,*)
```

```
JJJ = 1
```

Beginning of individual processor timer

```
IT1 = timer()
```

```
MM = M
NN = N
```

The following variables are initialized for use in the graphic routine RUNLENGTH WRITE.

```
XEND = 639
YEND = 479
BITS = 4
MULTR = 2**BITS
MAXIXC = INT(65535/MULTR)
SCREEN = (XEND+I)*(YEND+I)
```

```
H = L/MM
K = TI/NN
LAMB2 = (K*ALPHA/H)**2
```

Limiting the output to 50 iterations

```
NN = 50
```

This loop computes all of the boundary points for the vibrating string.

```
DO 6 J = 0, NN
   X = 0
   W(0,J) = SIN(3.1415927*0.)
   HOLDER(J,0,0) = INT(X*100+10)
   HOLDER(J,2,0) = INT(W(0,J))
   HOLDER(J,1,0) = HOLDER(J,2,0)+240
   HOLDER(J,3,0) = (YEND-HOLDER(J,1,0))*(XEND+1)
   + HOLDER(J,0,0)+1
```

```
The following two values are used in the pixel color computations.

\[
\text{HOLDER}(J, 0, MM+1) = \text{INT}((MM+1)*H*100+10)
\]

\[
\text{HOLDER}(J, 1, MM+1) = \text{HOLDER}(J, 1, MM)
\]

The initialization of the array associated with the counting semaphores for the completion of computations for the interior points for rows 0, ..., M-1

\[
\text{COUNT}(J) = 0
\]

This loop computes the initial conditions, the interior points for rows 0 and 1.

\[
\text{DO 20 II} = 1, \text{MM}-1
\]

Row j=0 computations

\[
\text{W}(II, 0) = \text{SIN}(3.1415927*II*H)
\]

\[
\text{HOLDER}(0, 0, II) = \text{INT}(x*100+10)
\]

\[
\text{HOLDER}(0, 2, II) = \text{INT} (\text{W}(II, 0))
\]

\[
\text{HOLDER}(0, 1, II) = \text{HOLDER}(0, 2, II)+240
\]

\[
\text{HOLDER}(0, 3, II) = (YEND-\text{HOLDER}(0, 1, II))*(XEND+I)
\]

20 CONTINUE

\[
\text{COUNT}(0) = \text{COUNT}(0) + 1
\]

25 CONTINUE

\[
J = JJJ
\]

\[
T = J * K
\]

\[
\text{DO 26 II} = 1, \text{MM}-1
\]

Row j=0 computations

\[
\text{W}(II, 1) = (1.-\text{LAMB2})*\text{W}(II, 0)
\]

\[
+ \text{LAMB2}/2.
\]

\[
+ \text{SIN}(3.1415927*(II+1)*H)
\]

\[
+ \text{SIN}(3.1415927*(II-1)*H))
\]

\[
+ K*0
\]

\[
\text{HOLDER}(1, 0, II) = \text{INT}(x*100+10)
\]

\[
\text{HOLDER}(1, 2, II) = \text{INT}(\text{W}(II, 1))
\]

\[
\text{HOLDER}(1, 1, II) = \text{HOLDER}(1, 2, II)+240
\]

\[
\text{HOLDER}(1, 3, II) = (YEND-\text{HOLDER}(1, 1, II))*(XEND+I)
\]

\[
+ \text{HOLDER}(1, 0, II)+1
\]

26 CONTINUE

COUNT(0) = COUNT(0) + 1
- \[ W(II, J-1) + \cos(2. \times 1.5927 \times T) \]

\[ + \frac{\sin((2. \times 3.1415927) \times T)}{2.71828182845 \times (10 \times T)} \]

\[ \text{HOLDER}(J+1, 0, II) = \text{INT}(X \times 100 + 10) \]

\[ \text{HOLDER}(J+1, 2, II) = \text{INT}(W(II, J+1)) \]

\[ \text{HOLDER}(J+1, 1, II) = \text{HOLDER}(J+1, 2, II) + 240 \]

\[ \text{HOLDER}(J+1, 3, II) = (Y \times \text{HOLDER}(J+1, 1, II)) \times (X + 1) + \text{HOLDER}(J+1, 0, II) + 1 \]

26 CONTINUE

\[ JJJ = JJJ + 1 \]

IF \((JJJ .NE. NN)\) GO TO 25

COUNT(J+1) = COUNT(J)

\(J = 0\)

Limiting the output to 50 iterations

\(NN = 50\)

33 CONTINUE

\[ \text{MAXIM} = -65535 \]

\[ \text{MINIM} = 65535 \]

Computations for pixel colors based on slope computations.

\[ \text{FLAG} = 0 \]

\[ \text{FLAG2} = 0 \]

\[ \text{FLAG3} = 0 \]

DO 35 I = 0, MM

IF \((\text{HOLDER}(J, 3, I) .GT. \text{MAXIM})\) \(\text{MAXIM} = \text{HOLDER}(J, 3, I)\)

IF \((\text{HOLDER}(J, 3, I) .LT. \text{MINIM})\) \(\text{MINIM} = \text{HOLDER}(J, 3, I)\)

SLOPE = \((\text{HOLDER}(J, 1, I+1) - \text{HOLDER}(J, 1, I))\)

IF \((0.0 .NE. \text{HOLDER}(J, 0, I+1) - \text{HOLDER}(J, 0, I))\) THEN

SLOPE = SLOPE / \((\text{HOLDER}(J, 0, I+1) - \text{HOLDER}(J, 0, I))\)

ELSE

SLOPE = 0.0

ENDIF

TEMP = ABS(SLOPE)

IF \(((0.0 .LE. \text{TEMP}) .AND. (\text{TEMP} .LT. 0.167))\) THEN

COLOR = 12

ELSE IF \(((0.167 .LE. \text{TEMP}) .AND. (\text{TEMP} .LT. 0.333))\) THEN

COLOR = 4

ELSE IF \(((0.333 .LE. \text{TEMP}) .AND. (\text{TEMP} .LT. 0.5))\) THEN

COLOR = 11

ELSE IF \(((0.5 .LE. \text{TEMP}) .AND. (\text{TEMP} .LT. 0.667))\) THEN

COLOR = 10

ELSE IF \(((0.667 .LE. \text{TEMP}) .AND. (\text{TEMP} .LT. 0.833))\) THEN

COLOR = 3

ELSE IF \(((0.833 .LE. \text{TEMP}) .AND. (\text{TEMP} .LT. 1.0))\) THEN

COLOR = 9

ELSE IF \(((1.0 .LE. \text{TEMP}) .AND. (\text{TEMP} .LT. 1.167))\) THEN

COLOR = 7

ELSE IF \(((1.167 .LE. \text{TEMP}) .AND. (\text{TEMP} .LT. 1.333))\) THEN

COLOR = 8

ELSE IF \(((1.333 .LE. \text{TEMP}) .AND. (\text{TEMP} .LT. 1.5))\) THEN

COLOR = 2

ELSE IF \(((1.5 .LE. \text{TEMP}) .AND. (\text{TEMP} .LT. 1.667))\) THEN

COLOR = 15
ELSE IF ((1.667.LE.TEMP).AND.(TEMP.LT.1.833)) THEN
  COLOR= 6
ELSE
  COLOR= 1
END IF

IF (SLOPE.GT.0.0) THEN
  CKSLOPE= 1
ELSE IF (SLOPE.EQ.0.0) THEN
  CKSLOPE= 0
ELSE
  CKSLOPE= -1
END IF

IF ((FLAG.EQ.0).AND.(SLOPE.NE.0.0)) THEN
  CKSLOPE= CKSLOPE
  FLAG= 1
END IF

IF (SLOPE.GT.0.0) THEN
  CKSLOPE= 1
ELSE IF (SLOPE.EQ.0.0) THEN
  CKSLOPE= 0
ELSE
  CKSLOPE= -1
END IF

CONTINUE

This section of the program is the inline encoding of
graphics routine, RUNLENGTH WRITE. This subroutine
loads color indices into the pixel viewport.

CONTINUE

IF (STOHOLD.LT.MINIMUM) THEN
  STRLINE(1:)= CHAR(27)
  STRLINE(2:)= CHAR(82)
  STRLEN= 3
  CALL DECCON(1)
  CALL DECCON(MULTR*MAXIXC+0)
  CODCOUN= CODCOUN+1
  LENGTH(J,CODCOUN) = STRLEN
  LINE(J,CODCOUN) (1:LENGTH(J,CODCOUN)) = STRLINE(1:STRLEN)
  STOHOLD= STOHOLD+MAXIXC
  GO TO 214
ELSE
  STRLINE(1:)= CHAR(27)
  STRLINE(2:)= CHAR(82)
STRLINE(3:) = CHAR(76)
STRLEN = 3

MINIMUM = MINIMUM - (STOHO LD - MAXIXC)

CALL DECCON(1)
CALL DECCON(MULTR*MINIMUM + 0)

CODCOUN = CODCOUN + 1
LENGTH(J, CODCOUN) = STRLEN
LINE(J, CODCOUN) (1:LENGTH(J, CODCOUN)) = STRLINE(1:STRLEN)

STOHO LD = STOHO LD + MINIMUM

INDXCOU = 0
END IF

DO 40 INDXPTR = MINIM, MAXIM
  DO 100 II = 0, MM
    IF (HOLDER(J, 3, II) .EQ. INDXPTR) THEN
      CK = 1
      IIHOLD = II
      GO TO 99
    END IF
  END DO 100
 CONTINUE
 CONTINUE

IF (CK .EQ. 1) THEN
  IF (INDXCOU .EQ. 0) GO TO 917
  STRLINE(1:) = CHAR(27)
  STRLINE(2:) = CHAR(82)
  STRLINE(3:) = CHAR(76)
  STRLEN = 3

  CALL DECCON(1)
  CALL DECCON(MULTR*INDXCOU + 0)

  CODCOUN = CODCOUN + 1
  LENGTH(J, CODCOUN) = STRLEN
  LINE(J, CODCOUN) (1:LENGTH(J, CODCOUN)) = STRLINE(1:STRLEN)

  CONTINUE

  STRLINE(1:) = CHAR(27)
  STRLINE(2:) = CHAR(82)
  STRLINE(3:) = CHAR(76)
  STRLEN = 3

  CALL DECCON(1)
  CALL DECCON(MULTR*1 + HOLDER(J, 4, IIHOLD))

  CODCOUN = CODCOUN + 1
  LENGTH(J, CODCOUN) = STRLEN
  LINE(J, CODCOUN) (1:LENGTH(J, CODCOUN)) = STRLINE(1:STRLEN)

  IN D XCOU = 0
  CK = 0
ELSE IF ((INDXCOU .EQ. MAXIXC) .OR. (INDXPTR .EQ. SCREEN)) THEN
  STRLINE(1:) = CHAR(27)
  STRLINE(2:) = CHAR(82)
  STRLINE(3:) = CHAR(76)
  STRLEN = 3

  CALL DECCON(1)
  CALL DECCON(MULTR*INDXCOU + 0)
CODCOUN= CODCOUN+1
LENGTH(J, CODCOUN) = STRLEN
LINE(J, CODCOUN) (1:LENGTH(J, CODCOUN)) = STRLINE(1:STRLEN)

INDXCOUN= 1
ELSE
  INDXCOUN= INDXCOUN+1
END IF

CONTINUE

MINIMUM= MAXIM+1
STOHOLD= STOHOLD+MINIMUM-MINIM

GO TO 444

IF (STOHOLD.LT.SCREEN) THEN
  STRLINE(1:)= CHAR(27)
  STRLINE(2:)= CHAR(82)
  STRLINE(3:)= CHAR(76)
  STRLEN= 3

  CALL DECCON(1)
  CALL DECCON(MULTR*MAXIXC+0)

  CODCOUN= CODCOUN+1
  LENGTH(J, CODCOUN) = STRLEN
  LINE(J, CODCOUN) (1:LENGTH(J, CODCOUN)) = STRLINE(1:STRLEN)
  STOHOLD= STOHOLD+MAXIXC

  GO TO 444
ELSE

  STRLINE(1:)= CHAR(27)
  STRLINE(2:)= CHAR(82)
  STRLINE(3:)= CHAR(76)
  STRLEN= 3

  CALL DECCON(1)
  CALL DECCON(MULTR*(SCREEN-(STOHOLD-MAXIXC)) +0)

  CODCOUN= CODCOUN+1
  LENGTH(J, CODCOUN) = STRLEN
  LINE(J, CODCOUN) (1:LENGTH(J, CODCOUN)) = STRLINE(1:STRLEN)

  INDXCOUN= 0
END IF

CODARY(J)= CODCOUN

CALL PXPOSIT(0,479)

DO 3 CODCOUN= 1, CODARY(J)
  WRITE(6,*), LINE(J, CODCOUN) (1:LENGTH(J, CODCOUN))
  CONTINUE

J= J + 1
IF (J.NE.NN+1) GO TO 33

IT2= timer()
TIME1(ME)= IT2-IT1

WRITE(6,*)
WRITE(6,*)
Barrier
TTEND= timer()
DO 3333 I = 1,NPROC
   WRITE(6,*) 'Processor ',I
   WRITE(6,*) 'Section time = ', TIMEI(I)
   WRITE(6,*)
CONTINUE

WRITE(6,'*)' The total time is ',(TTEND-TTBEG)
End barrier
Join

END

SUBROUTINE DECON(X)
C
C This graphics subroutine converts integer parameter
in host syntax.
C
COMMON DE,CON
CHARACTER *15 DE
INTEGER X,ABSNUM,DEC,CON
INTEGER BIN,HII,H12,LO1,H1IDEC,H12DEC,LO1DEC
DIMENSION BIN(0:15),HII(0:6),H12(0:6),LO1(0:6)
DIMENSION DEC(0:15)
C
C Initialization of arrays and local variables.
C
DO 5 K = 0,6
   H11(K) = 0
   H12(K) = 0
   LO1(K) = 0
5 CONTINUE
DO 10 K = 0,15
   BIN(K) = 0
   DEC(K) = 2**K
10 CONTINUE
H1IDEC = 0
H12DEC = 0
LO1DEC = 0
C
C Converts the INTEGER parameter to binary.
C
ABSNUM = IAABS(X)
DO 15 I = 15,0,-1
   IF (ABSNUM .GE. DEC(I)) THEN
      ABSNUM = ABSNUM - DEC(I)
      BIN(I) = 1
   ELSE IF (ABSNUM .EQ. 0) THEN
      GOTO 20
   ENDIF
15 CONTINUE
C
C Assigning bits.
C
20 H11(6) = 1
   H12(6) = 1
   LO1(6) = 0
   LO1(5) = 1
   DO 25 J = 0,5
      H11(J) = BIN(J+10)
      H12(J) = BIN(J+4)
      IF (J .LE. 3) THEN
         H12(J) = 1
LO1(J) = BIN(J)
ENDIF
CONTINUE

IF (X .GE. 0) THEN
LO1(4) = 1
ENDIF

Calculating the ASCII decimal equivalent (ADE) for array of bits.

DO 30 K = 0, 6
IF (HI1(K) .NE. 0) THEN
HI1DEC = HI1DEC + DEC(K)
ENDIF
IF (HI2(K) .NE. 0) THEN
HI2DEC = HI2DEC + DEC(K)
ENDIF
IF (LO1(K) .NE. 0) THEN
LO1DEC = LO1DEC + DEC(K)
ENDIF
CONTINUE

Transmitting the converted parameter to the terminal.

CON = CON + 1
DE(CON:) = CHAR(HI1DEC)
CON = CON + 1
DE(CON:) = CHAR(HI2DEC)
CON = CON + 1
DE(CON:) = CHAR(LO1DEC)
RETURN
END

SUBROUTINE XYCON(L,M)

This graphics subroutine converts xy-coordinates in host syntax.

COMMON PACK, NUM
CHARACTER*15 PACK
INTEGER NUM
INTEGER L,M, HIYDEC, EXTDCE, LOYDEC, HIXDEC
INTEGER LOXDEC, ABSNUM, DEC, XBIN, YBIN, EXTRA
INTEGER HIY, LOY, HIX, LOX
DIMENSION XBIN(0:II), YBIN(0:II), EXTRA(0:6)
DIMENSION HIY(0:6), LOY(0:6), HIX(0:6), LOX(0:6)
DIMENSION DEC(0:15)

Initialization of arrays and local variables.

DO 5 K = 0, 11
YBIN(K) = 0
XBIN(K) = 0
CONTINUE

DO 10 K = 0, 6
EXTRA(K) = 0
HIX(K) = 0
HIY(K) = 0
LOY(K) = 0
LOX(K) = 0
CONTINUE
DO 13 K = 0, 1
DEC(K) = 2**K
CONTINUE

HIYDEC = 0
EXTDEC = 0
LOYDEC = 0
HIXDEC = 0
LOXDEC = 0

CONVERTS THE INTEGER PARAMETERS TO BINARY.

ABSNUM = IABS(L)
DO 15 K = 1, 2
    DO 20 I = 11, 0, -1
        IF (ABSNUM .GE. DEC(I)) THEN
            ABSNUM = ABSNUM - DEC(I)
            IF (K .EQ. I) THEN
                XBIN(I) = 1
            ELSE
                YBIN(I) = 1
            ENDIF
        ELSE IF (ABSNUM .EQ. 0) THEN
            GOTO 25
        ENDIF
    CONTINUE
    ABSNUM = IABS(M)
    CONTINUE

ASSIGNING BITS.

HIY(6) = 0
HIY(5) = 1
EXTRA(6) = 1
EXTRA(5) = 1
EXTRA(4) = 0
EXTRA(3) = YBIN(1)
EXTRA(2) = YBIN(0)
EXTRA(1) = XBIN(1)
EXTRA(0) = XBIN(0)
LOY(6) = 1
LOY(5) = 1
HIX(6) = 0
HIX(5) = 1
LOX(6) = 1
LOX(5) = 0
DO 30 J = 0, 4
    HIY(J) = YBIN(J+7)
    LOY(J) = YBIN(J+2)
    HIX(J) = XBIN(J+7)
    LOX(J) = XBIN(J+2)
CONTINUE

CALCULATING THE ASCII DECIMAL EQUIVALENT (ADE) FOR ARRAY OF BITS.

DO 35 K = 0, 6
    IF (HIY(K) .NE. 0) THEN
        HIYDEC = HIYDEC + DEC(K)
    ENDIF
    IF (HIX(K) .NE. 0) THEN
        HIXDEC = HIXDEC + DEC(K)
    ENDIF
    IF (LOY(K) .NE. 0) THEN
        LOYDEC = LOYDEC + DEC(K)
    ENDIF
END IF
IF (LOX(K) .NE. 0) THEN
  LOXDEC = LOXDEC + DEC(K)
ENDIF

CONTINUE

Transmitting the converted parameter to the terminal.

NUM = NUM + 1
PACK(NUM:) = CHAR(HIYDEC)
NUM = NUM + 1
PACK(NUM:) = CHAR(EXTDEC)
NUM = NUM + 1
PACK(NUM:) = CHAR(LOYDEC)
NUM = NUM + 1
PACK(NUM:) = CHAR(HIXDEC)
NUM = NUM + 1
PACK(NUM:) = CHAR(LOXDEC)
RETURN

SUBROUTINE PXBEGIN(SURNUM, ALU, BPPIX)
This graphics subroutine sets up the terminal for subsequent pixel operations.

COMMON PX, BEG
CHARACTER *15 PX
INTEGER SURNUM, ALU, BPPIX, BEG

PX(I:) = CHAR (27)
PX(2:) = CHAR (82)
PX(3:) = CHAR (85)
BEG = 3
CALL DECCON(SURNUM)
CALL DECCON(ALU)
CALL DECCON(BPPIX)
WRITE (6, *) PX (I :BEG)
RETURN
END

SUBROUTINE PXPOSIT(XLOW, YLOW)
This graphics subroutine sets up the position of the pixel beam in the pixel viewport.

COMMON PX, POSIT
CHARACTER *15 PX
INTEGER XLOW, YLOW, POSIT
PX(I:) = CHAR (27)
PX(2:) = CHAR (82)
PX(3:) = CHAR (72)
POSIT = 3
CALL XYCON(XLOW, YLOW)
WRITE (6, *) PX (I :POSIT)
RETURN
END
SUBROUTINE PXVIEW(XLOW,YLOW,XHIGH,YHIGH)

This graphics subroutine specifies the pixel viewport's size and position in graphics memory.

COMMON PX, VIEW
CHARACTER *15 PX
INTEGER XLOW, YLOW, XHIGH, YHIGH, VIEW

PX(1:) = CHAR(27)
PX(2:) = CHAR(82)
PX(3:) = CHAR(83)
VIEW = 3
CALL XYCON(XLOW, YLOW)
CALL XYCON(XHIGH, YHIGH)
WRITE(6,*) PX(1:VIEW)
RETURN
END