Proceedings of the 1984
Winter Simulation Conference
S. Sheppard, U. Pooch, D. Pegden (eds.)

A MULTITASKING IMPLEMENTATION OF SYSTEM SIMULATION: THE EMULATION OF AN ASYNCHRONOUS PARALLEL PROCESSOR FOR SYSTEM SIMULATION USING A SINGLE PROCESSOR

Murali Krishnamurthi
Robert E. Young
Industrial Automation Laboratory
Dept. of Industrial Engineering
Texas A & M University

ABSTRACT

This paper describes the design and the development of a multitasking implementation of system simulation for the purpose of emulating an asynchronous parallel processor for system simulation using a single processor. The multitasking simulation system is a FORTRAN software kernel built around an existing simulation language (GASPIV) so that it emulates an asynchronous parallel processor simulation system. The multitasking simulation system is in full operation on a Texas Instruments 990/12 minicomputer and it is being used successfully to emulate the software architecture of the parallel processor simulation system being designed and built in the Industrial Automation Laboratory at Texas A & M University.

1.0 INTRODUCTION

The effectiveness of a newly developed software is an issue that requires serious consideration before an investment can be made on a special purpose hardware for that software. This requires testing and validating the effectiveness of the developed software. Testing by emulation is one of the best ways of testing a system's architecture. The main reason for implementing an existing simulation language using multitasking is to satisfy the need for an emulation system to test the software architecture of an asynchronous parallel processor simulation system. The multitasking simulation system described in this paper is a FORTRAN software kernel built around the GASPIV simulation language and it is implemented on a Texas Instruments 990/12 minicomputer. The implemented multitasking system has proved to be an efficient emulation system for the asynchronous parallel processor simulation system being designed and built in the Industrial Automation Laboratory at Texas A & M University.

Section 2.0 presents an overview of multitasking and the differences between multitasking and parallel processing. A brief description of the multitasking features available for high level languages has been included in this section. Section 3.0 discusses the design and the development of the software for the multitasking simulation system. Section 4.0 describes the implementation and the testing of the system and Section 5.0 presents the summary of the paper and the results.

2.0 THE MULTITASKING CONCEPT

2.1 Definition

Before defining multitasking and explaining its features, certain related terms should be defined. A "Task" is defined as a complete logical execution path through a program demanding the use of system resources such as the CPU and I/O device control. Any computer program we normally run is a single task. "Time Slicing" refers to the sharing of CPU time among several tasks alternately giving each a short interval (time slice) of time [HOLT78]. "Multitasking" is a computer resource management approach which allows more than one user's program (task) to execute using time slicing. This gives the appearance that each program owns the CPU.

2.2 Task States and Execution Structure

When executing a set of tasks using multitasking, only one task at a time can own the CPU and be executing. The other tasks are maintained in either a "READY," "SUSPENDED," or "DORMANT" non-executing state. The READY state is used for tasks which are available for execution but cannot gain control of the CPU until all higher priority tasks existing in the READY or EXECUTING state have completed or SUSPENDED. A task is put in the SUSPENDED state if it is awaiting the occurrence or completion of an operating system function or real-time data from an external environment. A task is put in the DORMANT state if it has not been introduced into the "active" job set or if its execution has been completed. Figure 1 illustrates the state transitions for a typical task. It can be seen from the figure that a task can exist in only one state at any given time. For a task to be executed, it must proceed through the prescribed logical, sequence displayed in Figure 1. For example, a task in the DORMANT state cannot be directly SUSPENDED. Similarly, a DORMANT task has to be brought to the READY state before it can be SUSPENDED.

2.3 Advantages of Multitasking

The multitasking feature is often required in a computer system for efficient use of hardware resources and to provide quick response to user's requests. "Hardware inefficiencies are typically due to the fact that system's hardware resources run in parallel at vastly different speeds ...." [HOLT78]. For example, the time taken by a console to process a single character may vary from a tenth

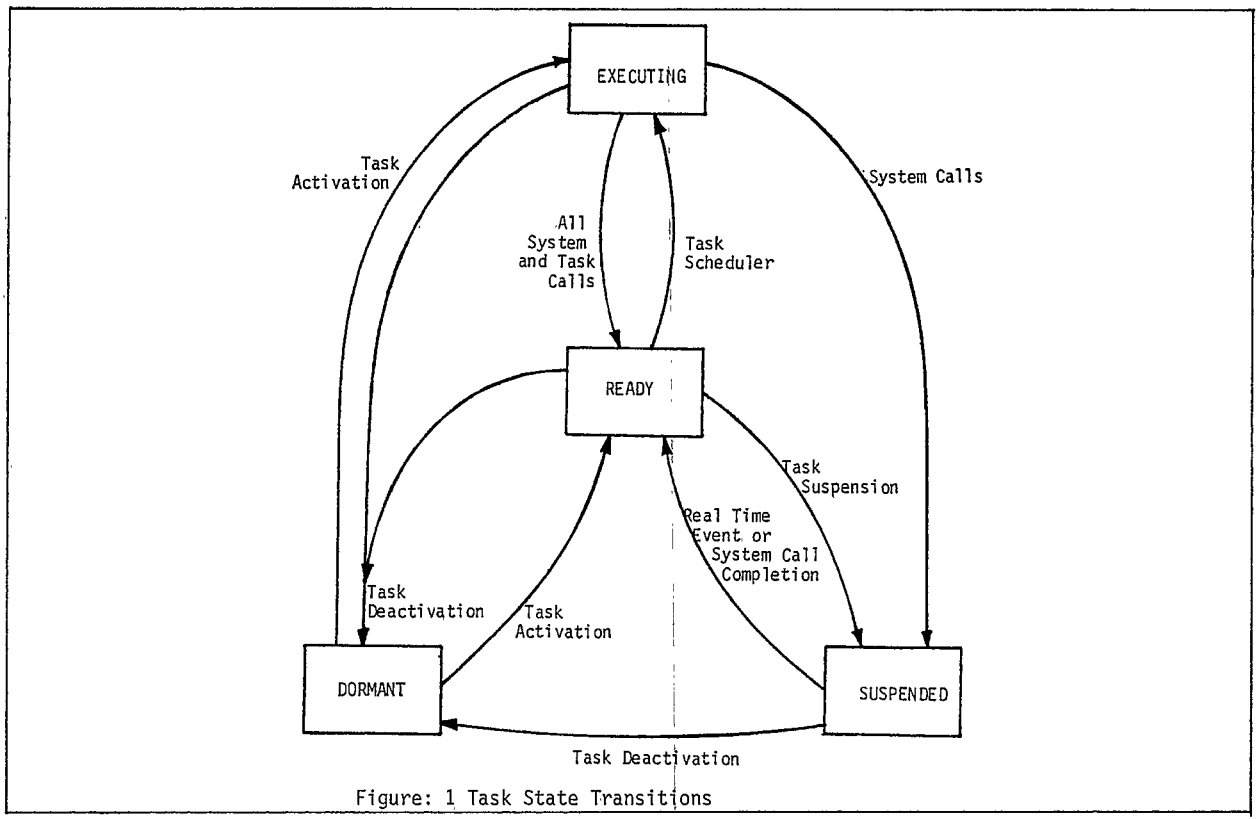

Figure: 1 Task State Transitions

of a second compared to a thousandth of a second for a card reader and a millionth of a second for a CPU. Obviously the CPU will be wasting its time if its idle while a console is transmitting a character. While a user is typing messages to a running task, another task should be given the CPU. If a job is I/O bound, spending most of its time waiting for input/output devices, the spare CPU time can be used by a compute bound job, which spends most of its time using the CPU. This parallel mode of operation can be accomplished by multitasking.

Apart from increasing the efficient use of hardware resources, multitasking allows the computer system to respond quickly to user's needs. This phenomenon is illustrated in Figure 2. The difference between the throughputs for executing 3 tasks in a single task operating system and the same in a multitasking operating system can be clearly seen. It should be understood here that the mentioned facts are true also for multitasking operating systems without high level language access to the multitasking features. Another advantage of multitasking is the increase in memory allotted to a program when it is run in a multitasking environment as a collection of tasks rather than as a single program unit. Computer systems such as the TI 990/12 and the PDP 11/xxx series have limitations on memory allotted to task execution. If the same program can be broken down into several smaller tasks, each smaller task will get the same amount of memory for execution as the original program ran as a single task. This may be of significant advantage for executing programs which require more memory than they are allocated for execution.

2.4 Multitasking and Parallel Processing

As mentioned in Section 1.0 the primary objective of this research effort is to design and develop a multitasking system that will emulate an asynchronous parallel processor for system simulation. Let us now consider the differences between multitasking and parallel processing and the reasons for choosing a multitasking system to emulate a parallel processor system. The main difference between multitasking and parallel processing is in the hardware configuration. Multitasking uses one processor for all tasks and it will use the same amount of processing time as a non-multitasking system. Multitasking will reduce the flow time of tasks in the system due to time slicing and prioritized scheduling of tasks. The parallel processing system is much superior to the multitasking system since the tasks can be overlapped and more than one processor can be executed at the same time. This overlapping of the tasks results in better throughput compared to the multitasking system. In the case of pure sorting and searching type problems the computation time may be speeded up by N times using N processors [JONE80].

Another major difference between multitasking and parallel processing is in the software architecture of the two systems. In an ideal situation, the parallel processing system tasks will be strictly independent of each other. In practice few problems exhibit such an ideal structure. The required synchronization in a parallel processor hardware environment is accomplished by messages

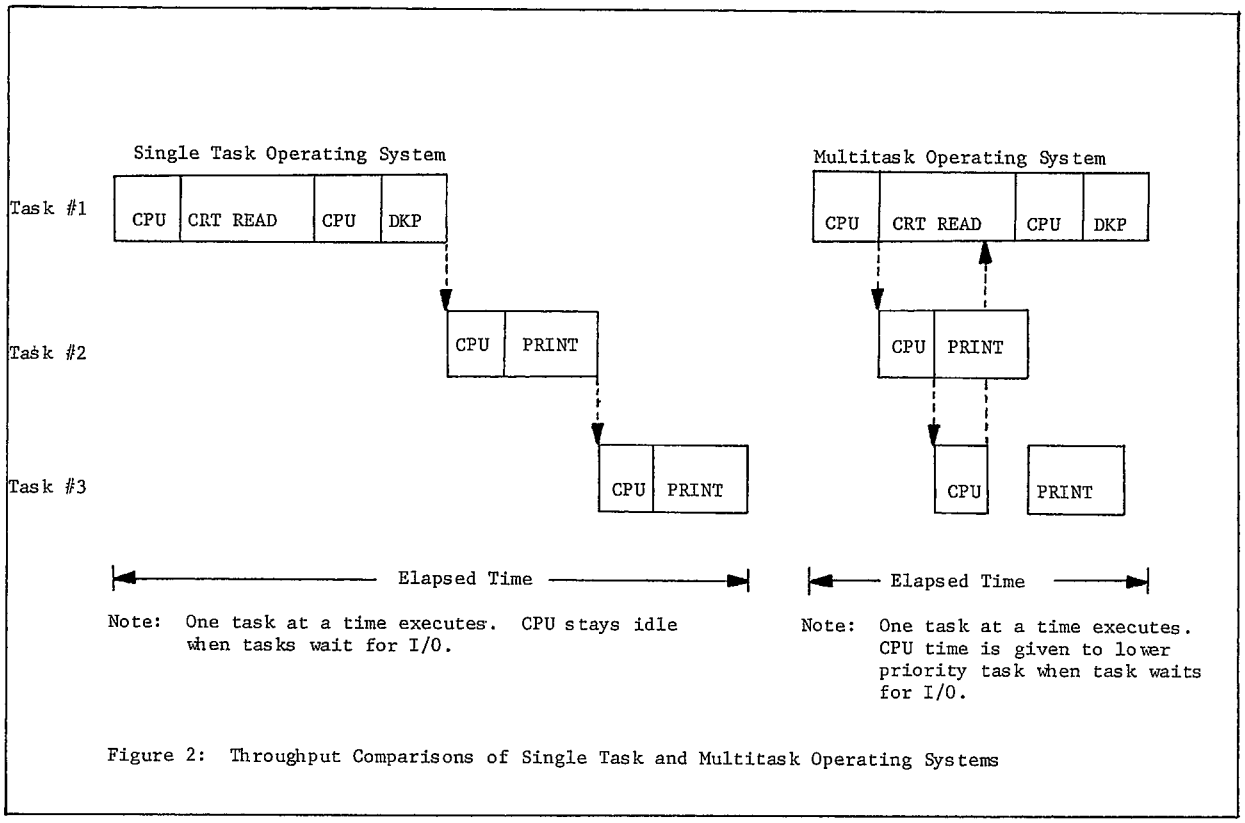

Figure 2: Throughput Comparisons of Single Task and Multitask Operating Systems

passed through memory and not through software calls (except in Ada in which parallel tasks on separate processors can communicate via a "rendezvous" which is a software call). The multitasking system tasks can be synchronized both by messages passed through memory and by software calls. Also in a multitasking system only one task can be executing at any given time since there is only one processor. In a parallel processing system it is possible for several tasks to execute at the same time and access or update the same data which may result in a "dead lock" situation. In spite of the differences between a multitasking and a parallel processing system we have chosen the multitasking system to emulate the parallel processing system for the following reasons:

1. The multitasking configuration is the only configuration that comes reasonably close to the parallel processing system since both systems have multiple tasks interacting with each other.
2. The software developed for the parallel processing system can be tested on the multitasking system with minor modifications to suit the multitasking system and decisions can be made with respect to the effectiveness of various software design alternatives without the need for the hardware changes required for the parallel processing system.

2.5 TI 990/12 DX 10 Architecture

For the purpose of implementing the multitasking simulation system we chose to use the Texas Instruments 990/12 minicomputer located in the Industrial Automation Laboratory at Texas A & M University. The TI 990/12 is a 16-bit minicomputer with 576K bytes of memory. The system runs under the DX 10 operating system which has excellent multitasking capabilities. The system is configured to allow as many as 8 tasks to execute at any given time. Each task is limited to approximately 64K bytes. The multitasking features on the TI 990/12 are also accessible from high level languages such as FORTRAN and Pascal. The reader is referred to [TIOS82] for more information on the TI 990/12 and its DX 10 operating system.

2.6 Task Communication Methods in TI 990/12

Since the tasks in the parallel processing system had to be synchronized, we needed to choose a way to allow the tasks to communicate in the multitasking emulation which represented the way the parallel processing system would operate. The types of communication between tasks are usually the passing of messages and the passing of parameters. Messages are "hand shaking" signals between tasks to inform one another of task status. For example, a task can set its task status variable to executing, suspended or dormant so that other tasks can synchronize themselves with this task. The parameter passing communication between tasks serves both as message and data necessary for task synchronization. The TI 990/12 allows both types of communication through Supervisory Calls (SVC) from high level languages. SVCs are predefined routines in the operating system that perform functions generally required by assembly language programs such as opening files, assigning logical units, and so on. The TI 990/12 DX 10 operating system has 70 SVCs [TISV82] that provide program support, file I/O and device I/O. These

SVC calls facilitate the activation, reactivation, suspension, self-suspension, delay, termination, self-identification and several other functions needed for task interaction. Tasks can communicate with each other by one of the following four methods: (1) through a synonym table, (2) through an SVC call block, (3) by writing to an I/O file and (4) through a global common data area. The simplest of the mentioned methods is communicating through a global common data area since operating system functions are not required for this type of communication. We have chosen global common for communication for our system for the mentioned reason. It should be noted that that in newer languages (particularly ADA [ANSI83]) many of these features are directly implemented in the high level language itself. This promises to simplify the development and the maintainability of the multitasking software.

2.7 Multitasking in FORTRAN-78 on the TI 990/12

We chose to develop our multitasking system in FORTRAN-78 [TIFT82] which is available on the TI990/12. FORTRAN-78 is an extended FORTRAN-77 with additional features pertaining to the TI 990/12 system. FORTRAN-78 allows the calling of earlier mentioned operating system subroutines through a simple "CALL SVC" statement. Also there exist FORTRAN-78 library subroutines such as EXTASK (execute a task), START (start a task) and WAIT (delay the execution of a task) which can be directly called from FORTRAN. The simplest of the approaches for passing parameters in FORTRAN-78 is through a global common data area. In FORTRAN, a global common data area is established through a BLOCK DATA subprogram. This BLOCK DATA subprogram must be declared as a "shared procedure" among the tasks and the tasks can communicate with each other through the variables and data structures declared in the global common block. To implement the data sharing, the tasks are compiled separately and then linked with this "shared procedure." The linked object modules of individual tasks are combined together to form a program file (load module) for the multitasking program. This method of compiling and linking subprograms allows the individual tasks to communicate with each other when the program file is loaded for execution since the tasks are in the same program file. The reader is referred to [TIFT83] for more information on programming and linking in FORTRAN-78 and the multitasking capabilities available in FORTRAN-78. Similar capabilities are also available in Pascal.

The multitasking concept and the high level multitasking features available on the TI 990/12 have enabled us to design an efficient emulation system for the parallel processing simulation system. The next two sections describe the design and the implementation of the multitasking simulation system.

3.0 DESIGN OF THE MULTITASKING SIMULATION SYSTEM

3.1 Design Rationale Behind the Multitasking Simulation System

In many traditional simulation systems, the simulation analysis program (simulation package) is organized as "one big program" or task that is executed just like any other program on the computer. But in a multitasking simulation system, the analysis program can be divided into several tasks that perform the individual simulation language functions. These tasks can be supported by task drivers and task interface subroutines that enable the tasks to interact with each other.

One approach to developing the mentioned system would be to completely redesign and restructure the basic simulation analysis functions for a multitasking environment. However we had established as design goals: (1) not to change a simulation language's existing execution structure and (2) not to modify the existing user interface so that user will not have relearn to use the language. These goals essentially stipulate that the multitasking implementation should be user transparent and also provide an emulation of the parallel processor simulation system.

3.2 Functional Breakdown of GASPIV to Facilitate Multitasking

As our target simulation system we decided to use the GASPIV Simulation Language [PRIT74] in our multitasking simulation system. The reader is referred to [YOUN84] for more information regarding the reasons for choosing GASPIV and the design rationale behind the parallel processing simulation system. Since it was our intent to emulate the parallel processor system, the same design rationale was used in the design of the multitasking system.

The first step in the design process required the analysis of the GASPIV system to identify major functional groups which could be shown to demonstrate relative independence during execution. GASPIV simulation language functions were grouped into eight compute tasks as follows:

1. Data input and system initialization
2. Simulation supervision and event sequencing
3. "User Event" software
4. Filing system management
5. Statistical collection management
6. Random number generation
7. Simulation output analysis and reporting
8. Numerical integration and Continuous variable management

(These tasks will be referred to as INPUT, SUPERVISOR, USER, FILING, STATISTICS, RANDGEN, OUTPUT and CONTINUOUS respectively for reference in the following discussion). We designated the "USER" task to include all routines which do not fall into one of the other categories. Figure 3 shows the eight partitions and the GASPIV subprograms grouped within each partition.

3.3 Design Considerations for a Multitasking Simulation System

Unlike other popular simulation environments such as SLAM [PRIT79], SIMAN [PEGD83] or IDSS 2.0 [USAF83], GASPIV only provides a program framework and a basic set of simulation utilities. This makes GASPIV very flexible and powerful but provides as well some unique problems for the multitasking environment. In GASPIV the user writes the main program, event routines, system initialization routines and other necessary routines. When the user's main program is

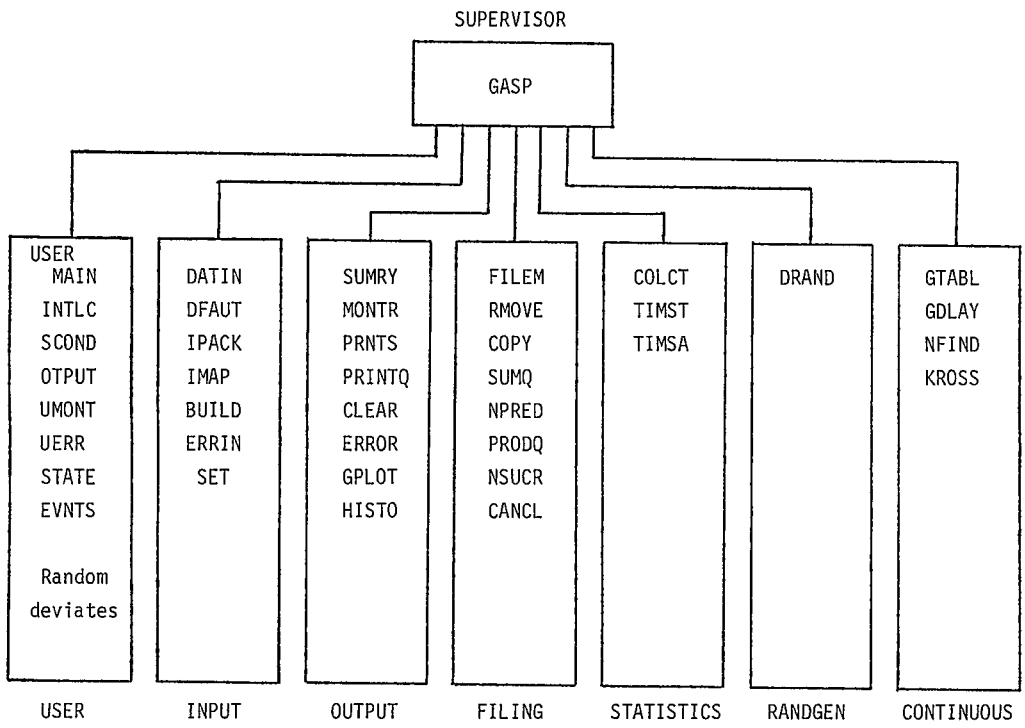

Figure 3. GASPIV Subprogram partition groups

executed, it calls subroutine GASP which establishes the simulation environment. Subroutine GASP calls the input routines to read the input data and initialize the system as well as to enter the initial events into the event list. Subroutine GASP then repeatedly checks the event list and calls the appropriate user events till the specified completion time of simulation. In the user event routines, the next events are generated and filed in the event list and the necessary computations are made. After simulation is completed, subroutine GASP calls the output and statistical routines to output the results. Subroutine GASP then returns to the user's main program and quits.

As mentioned above one of the primary design criteria for the multitasking system was to maintain the functional flow of GASPIV. The partitioning of the simulation language functions into separate tasks to emulate the parallel processing environment required the resolution of the following issues in order to achieve the mentioned aspect of user transparency:

- How to call a subroutine in one task from another task
- How to pass parameters between tasks and subroutines
- How to call user subroutines from another task
- How to terminate the tasks automatically after the completion of the simulation
- How to report multitasking errors
- How to find the status of a task (whether the task is executing or dormant)
- How to read and write to a user file from a task

These design issues are discussed in the next two sections and appropriate solutions are provided.

3.4 Design of the Multitasking Interface Layer

To solve these problems we designed a special global common block (BLOCK DATA subprogram) which serves as the communication path between the tasks and the subroutines. The global common block contains the GASPIV common blocks GCOM1, GCOM2, GCOM3, GCOM4, GCOM5, GCOM6 and and GCOM7 which are needed by most of the GASPIV subroutines. GCOM7 is not an original GASPIV common, rather it is the unnamed common containing QSET that has been modified to a named common since unnamed common blocks cannot be declared in a BLOCK DATA subprogram. The global common also contains common blocks that declare task and subroutine monitoring variables and the subroutine parameters. The monitoring variables are semaphores used to represent the status of a task or a subroutine. If a task or subroutine is in execution, its corresponding monitoring variable is set. Once the task has finished its execution the monitoring variable is reset. For example, if Task 1 has to execute Task 2, first Task 1 will check the task monitoring variable of Task 2 to see if Task 2 is executing or idle. If Task 2 is idle, Task 2's monitoring variable will be set and activated. Once Task 2 has completed its execution, its monitoring variable will be reset. The monitoring variables for all the tasks are declared in a common block. The monitoring variable for each subroutine is declared in separate common blocks.

The parameters for the subroutines are declared in the same common block as its monitoring variable.

For the sake of uniformity and to prevent the users from using in their named common blocks the same labels used in the global common, a uniform coding scheme has been used to define the common block names for tasks, subroutines and functions. The names of the common blocks declaring tasks, subroutines and functions have been defined to begin with T, S, and F respectively. For example, the name of the common block declaring task INPUT will be "TINPUT", the name of the common block declaring subroutine FILEM will be "SFILEM" and the name of the common block declaring the function SUMQ will be "FSUMQ." Similarly, the monitoring variables and the returned values of functions have been uniformly defined to begin with "M" and "V" respectively. For example, the complete common block declaration of subroutine FILEM will be "COMMON /SFILEM/ MFILEM, IFFILE" where IFFILE is the parameter passed to the subroutine FILEM. The second major design goal mentioned was to minimize the changes to the user program-interface. Simply stated we did not want the user to be burdened with the responsibility of managing the inter-task coordination. Since we were pursuing performance and not functional improvements we wanted to completely isolate the user from any knowledge of the implementation. We also wanted to execute existing simulation language programs with little or no change. Because of the fact that the GASPIV users are actually writing FORTRAN programs while they are constructing their models, the achievement of the earlier mentioned goal proved to be very difficult. The final solution required the establishment of a software layer between the user/programmer and the actual analysis software as shown in Figure 4. This layering approach is identical in concept to that being used by several research systems addressing information in heterogenous distributed database environments [McLE83] and [SMIT81]. Our layer was comprised of two major components, "the task drivers" and "the task interface library."

The task driver is the main program of a task that contains the global common declarations of all subroutines that are referenced outside the task. The task driver can execute another task or self-suspend indefinitely anytime during the active state. The task driver together with the subroutines from a functional group form a particular task. The task driver is the mechanism used to execute a subroutine residing in that task when requested by a subroutine in the "task interface library." Figure 5 shows the task driver for "FILING."

The Task Interface Library (TIL) consists of "shell subroutines" of all subroutines residing in each task that are referenced by other tasks outside the resident task. Thus the shell subroutine is the communication link between a calling subroutine and the subroutine in another task that is being called. When a task requires a subroutine and finds that the required subroutine is not residing in the same task, the task interface library is referenced and the required subroutine is called. This subroutine in the TIL is a shell subroutine version of the actual subroutine having the same name and calling parameters. When the shell subroutine is called it sets its monitoring

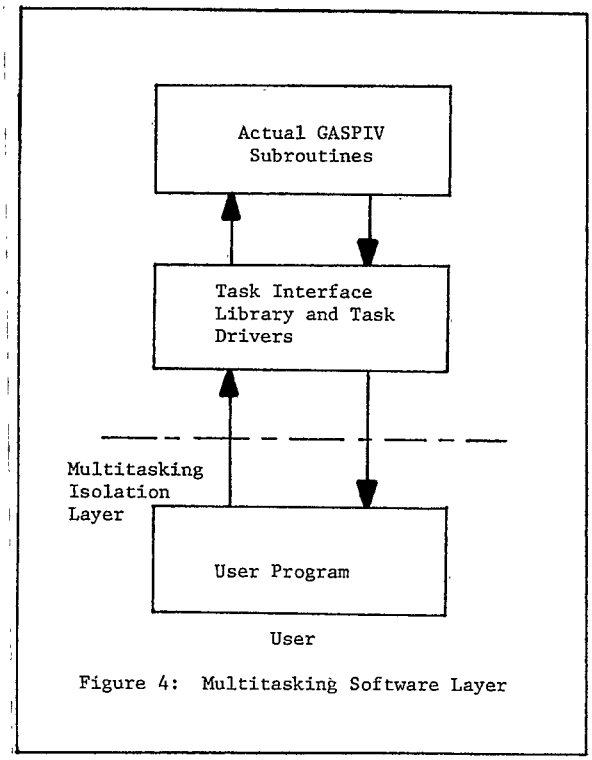

Figure 4: Multitasking Software Layer

variable in global common and passes the required subroutine parameter to the corresponding variable in global common. Then the shell subroutine checks if the task containing the actual subroutine is executing or dormant. If the task is active, the shell subroutine waits till the task becomes dormant and reactivates that task. Then the shell subroutine keeps checking the task monitoring variable to see if the task has become dormant again. The reactivated task checks its subroutine monitoring variables to see which subroutine to execute and calls the required subroutine and passes the subroutine parameters from global common. After completion the subroutine returns back to the task driver. The task driver then resets the task monitoring variable in global common to dormant and self-suspends indefinitely. The shell subroutine notes this status change in the task monitoring variable and returns back to the called subroutine. Thus the shell subroutine serves as an indirect mechanism for referencing a subroutine that is not residing in the calling task. The concept and the operation of the interface layer as implemented with the shell subroutines are illustrated using the shell subroutine for subroutine FILEM in Figure 6.

To establish the requirements for the interface layer we analyzed the call relationships between the 8 functional groups. This analysis indicated a need for 33 shell subroutines. The shell subroutines are logically and functionally similar to one another since they all serve the same general purpose. In fact all the shell subroutines (except DRAND) differ only in the task they activate and the data they update in the global common area.

```
C*****Program main (Task driver) for task FILNG
C*****Declare task monitoring variables of all the
C*****tasks, in global common.
C
      COMMON/TSKSTA/ MUSER,MSUPER,MINPUT,MOUPUT,
     &MFILNG,MSTATS,MRNDGN,MCONTN
C
C*****Declare in global common, the subroutine
C*****monitoring variables and the parameters of
C*****all the subroutines residing in task FILNG
C
      COMMON /SFILEM/ MFILEM,IFFILE
      COMMON /SRMOVE/ MRMOVE,INTRY,IRFILE
                . .
                . .
      COMMON /FSUMQ/ MSUMQ,VSUMQ,JSATT,ISFILE
C
C*****Define SVC CALL parameters
C
      DIMENSION SVCBLK(18),REGBLK(10)
      DATA SVCBLK/18*0/
C
C*****Reset task FILNG monitoring variable to free
C*****(i.e. MFILNG=0) and self-suspend indefinitely
C*****using the "self-suspend SVC" call.  SVCBLK(1)
C*****represents the opcode for self-suspend SVC.
C
  10  MFILNG    = 0
      SVCBLK(1) = >0600
      CALL SVC(SVCBLK,IERR,.FALSE.,REGBLK)
C
C*****If any multitasking error occurrs during the
C*****self-suspension of task, report the error
C*****to subroutine SVCERR
C
      IF(IERR.NE.0)CALL SVCERR(951,IERR)
C
C*****Task reactivated by a shell subroutine or by
C*****the USER task. Reset task monitoring variable
C*****to active (i.e. MFILNG=1).
C
      MFILNG = 1
C
C*****Find which shell subroutine reactivated this
C*****task by checking the subroutine monitoring
C*****variables of all the subroutines residing in
C*****this task.  Call the subroutine which
C*****reactivated this task and pass the required
C*****for the subroutine through global common.
C
      IF (MFILEM.EQ.1) THEN
         CALL FILEM(IFFILE)
         MFILEM = 0
      ENDIF
           . .
           . .
      IF(MSUMQ.EQ.1) THEN
        VSUMQ = SUMQ(JSATT,ISFILE)
        MSUMQ = 0
      ENDIF
C
C*****Check if task SUPERVISOR is still active. If
C*****it is not (end of simulation) terminate this
C*****task.  Otherwise. self-suspend indefinitely.
C
      IF (MSUPER.EQ.1) GOTO 10
      STOP
      END
```

Figure 5. Task Driver for task FILING

```
C*****Shell Subroutine FILEM
C
      SUBROUTINE FILEM(IFILE)
      INTEGER*2 STATID, TASK
C
C*****Declare the task monitoring variables of all
C*****the tasks, in global common
C
      COMMON/TSKSTA/ MUSER,MSUPER,MINPUT,MOUPUT,
     &MFILNG,MSTATS,MRNDGN,MCONTN
C
C*****Declare the monitoring variable for
C*****subroutine FILEM and its parameter,
C*****in global common
C
      COMMON /SFILEM/ MFILEM, IFFILE
C
C*****Define SVC call parameters. LUNO represents
C*****the logical unit number assigned to the task
C*****and TASK(1) represents task identification
C*****number in hexadecimal
C
      DIMENSION TASK(3)
      DATA STATID/0/
      LUNO    = >FF
      TASK(1) = >4
C
C*****Check if task FILNG is free (i.e. MFILNG=0).
C*****If yes, put the parameters of subroutine
C*****FILEM in global common and set subroutine
C*****FILEM and task FILNG monitoring variables
C*****to active (i.e. MFILEM=1, MFILNG=1).
C
  5   IF (MFILNG.EQ.0)THEN
         MFILNG = 1
         IFFILE = IFILE
         MFILEM = 1
C
C*****Reactivate task FILNG using the high level
C*****multitasking call EXTASK and if there are
C*****any multitasking errors (i.e. IERR.NE.0)
C*****report them to subroutine SVCERR.
C
         CALL EXTASK(LUNO,TASK,STATID,.FALSE.,IERR)
         IF(IERR.NE.0)CALL SVCERR(1401,IERR)
C
C*****Task FILNG is active, wait for 50 millisecs.
C*****using the high level time delay subroutine
C*****WAIT and check the status of FILNG again.
C
      ELSE
         CALL WAIT(50,1,ISTAT)
         GOTO 5
      ENDIF
C
C*****Check if task FILNG has finished executing
C*****SUBROUTINE FILEM (i.e. MFILNG=0).  If yes,
C*****return to the calling task.  Otherwise,
C*****wait for 50 milliseconds and check the
C*****status of task FILNG again.
C
  10  IF (MFILNG.EQ.0)THEN
         RETURN
      ELSE
         CALL WAIT(50,1,ISTAT)
         GOTO 10
      ENDIF
      END
```

Figure 6. Shell subroutine for Subroutine FILEM

Parallel processing requires the tasks be mutually exclusive in terms of processing activities and also in terms of results. An exception to this is random deviate generation. The random deviate generators and the user software have a high degree of result inter-dependence. Random deviate generation requires passing parameters to a random deviate generator program. We have placed the random deviate generation subroutines in the USER task so that the result inter-dependence is constrained to a single task grouping [YOUN84]. The random numbers can be generated in advance and referenced by the random deviate generators as needed. This implementation approach was followed because it truly emulated the generation of random numbers in parallel. This required the modification of the shell subroutine for DRAND (the random number generator subroutine). The first time the shell subroutine for DRAND is called, it calls the random number generator task, initializes the seed, generates a random number and returns the value to the calling task. Then it also generates a second random number and saves it in global common. On such subsequent call when the DRAND shell subroutine is executed, it returns immediately to the calling task the random number generated in the previous call. This allows the interleaving of the calling subroutine during the generation of the next random number which is saved in global common.

The design of the USER task is slightly complicated due to the unique functional flow of GASPIV. In the traditional implementation of GASPIV, the user main program directly calls subroutine GASP. But in the multitasking system we have subroutine GASP in the SUPERVISOR task and the user main program cannot directly call it. Also, the USER task cannot have a task driver as its main program just like the other tasks since the user writes its main program. In order to maintain the existing interface between user main program and subroutine GASP, we need a task driver for the USER task to enable other tasks to call user subroutines such as INTLC, OTPUT and STATE. We accomplished this by naming the USER task driver as GASP. This task driver was designed as a subroutine to reside in the USER task. When the user main program calls subroutine GASP, the task driver which is named GASP is executed. This task driver then activates the SUPERVISOR task and executes the actual subroutine GASP.

The above sections have discussed the major operational and structural features of our implementation. However, we still have to look at some of the other design requirements such as the automatic termination of a task and reporting of multitasking errors. First, let us consider terminating a task automatically. When a task other than the SUPERVISOR task is executed, it self-suspends after executing a required subroutine residing in it. If the task is terminated every time after its execution instead of self-suspension, the task will lose the updated data on all its variables and will have only the initial values of all variables. An example of the possible effect this lost data could have is best illustrated by the random number generator task. After activating the random number task and generating the random number, if the task is terminated, the same random number will be generated every time the random number generator task is activated since the updated seed will not be saved due to the completion of the task. Instead, if the task is allowed to self-suspend but not terminate after generating a random number, the updated seed will be saved and proper random numbers will be generated during succeeding task activations. On the other hand, we need the tasks to terminate automatically at the end of a simulation run. Thus we need a global semaphore that indicates the status of the simulation.

We see in the functional flow of GASPIV that the subroutine GASP is the first one to be executed among the GASPIV subroutines and it is also the last one to finish execution. Subroutine GASP acts as the supervisor for the entire simulation and its completion indicates the end of the simulation. We have used the status of the SUPERVISOR task to indicate to the other tasks the status of the simulation run since the SUPERVISOR task is active throughout the simulation and is never self-suspended. At the end of a simulation run, the SUPERVISOR task resets its monitoring variable to inactive, reactivates the USER task and self-terminates. The USER task reactivates other tasks before returning to user's main program. The reactivated tasks, before self-suspending, check the status monitoring variable of task SUPERVISOR and if the task is still active, the tasks self-suspend otherwise terminate themselves by branching to STOP and END statements.

3.5 Design of the Error Monitoring Features for the Multitasking Simulation System

The chances of multitasking errors occurring in the simulation is negligible. The multitasking simulation system is user transparent and once the system has been fully debugged and properly implemented, there is little possibility of software errors occurring in the system. In spite of this fact error reporting routines have been added in all tasks to report exactly where and what type of a multitasking error has occurred since it is possible for the operating system errors or hardware failures to affect the multitasking simulation system.

3.6 Modifications Made in GASPIV to Implement it in the Multitasking Simulation System

One major difference between the traditional implementation of GASPIV and the multitasking implementation of GASPIV is in the I/O operations. Since the tasks in a multitasking simulation program are separate programs under the TI 990/12 DX 10 operating system, the tasks cannot access the same file for reading or writing purposes. In the traditional GASPIV implementation, the entire simulation output is written into the same file. In the multitasking implementation, each task has its own output file requiring the collation of the output from all tasks at the end of the simulation run. The output collation is made user transparent by using an operating system procedure.

Once the design of the multitasking simulation system had been completed, the next step was to develop the software. The software development required identifying the appropriate multitasking calls and their parameters required. The multitasking system required only the following three types of multitasking calls:

CALL SVC - for self-identification, self-

suspension of a task and reactivation of a suspended task
CALL SVCFUT - for opening a disk file to READ or WRITE from a task
CALL EXTASK - for executing a task (this is a FORTRAN-78 library function)

These calls required parameters such as the operation code (whether self-identification or self-suspension etc.,), logical unit number to which the task is attached, the task identification number (this is obtained during program implementation) and an error monitoring variable. Using these multitasking calls and the design described so far, the task drivers and the pseudo subroutines were developed (see Figures 5 and 6).

4.0 IMPLEMENTATION OF THE MULTITASKING SYSTEM

4.1 Installation of the Multitasking Simulation System

System implementation consisted of compiling, debugging, linking, installing and testing the software developed. Linking multitasking software is quite different from linking conventional, single task software. A multitasking system requires each task to be linked separately to the necessary libraries and shared procedures. For the multitasking simulation system, we linked each task (except the USER task) separately to the task interface library, other necessary utility libraries and the BLOCK DATA subprogram. The BLOCK DATA subprogram was linked as a shared procedure among all tasks so that it can be used for communication between the tasks.

The linking and the installation of the USER task has to be done by the user. The necessary linker control stream (similar to JCL used for linking in an IBM mainframe computer) for the USER task is provided in the TI 990/12 system so that users can link their main program and other needed subroutines to the USER task. The USER task is linked with user programs, USER task driver (subroutine named GASP), shared global common (BLOCK DATA subprogram), the task interface library and the other necessary libraries. The linked tasks are then installed in a program file [TIOS82]. The user is required to install the USER task in the same program file. A program file consists of executable linked outputs of all tasks along with their shared procedure. Each task is installed in the program file with a task identification number. The task identification is used in the system to activate, self-suspend or delay a task.

4.2 Execution Structure of the Multitasking Simulation System

Once the user has installed the USER task and created all the necessary input and output files, the simulation can be started by executing the USER task just like any other FORTRAN program on the computer. When the USER task is executed, the user main program executes first and calls subroutine GASP residing in the same task. The subroutine GASP (task driver of the USER task) first activates all the tasks except the SUPERVISOR task. The activated tasks check the subroutine monitoring variables of all the subroutines residing in their task, find that none of them have to be called and self-suspend. The USER task driver then activates the SUPERVISOR task and self-suspends indefinitely. The activated SUPERVISOR task calls the actual subroutine GASP residing in the same task and from then on subroutine GASP takes over as in the conventional GASPIV simulation analysis program.

During the simulation, if a subroutine calls another subroutine which is not residing in the same task, the shell subroutine of the called subroutine in the task interface library will be referenced. The shell subroutine will transfer the subroutine call parameters to global common, set the subroutine monitoring variable and activate the task in which the actual subroutine is residing. The activated task will check the subroutine monitoring variables of all the subroutines residing in it and will find the appropriate subroutine to call. The task driver will get the subroutine parameters from global common and call the required subroutine. Once the subroutine has completed its execution, the task driver will reset the subroutine monitoring variable and the task monitoring variable to idle. Then it will check the task monitoring variable for task SUPERVISOR and self-suspend indefinitely if the SUPERVISOR is still active or terminate itself if the SUPERVISOR has completed its execution.

At the end of the simulation, the SUPERVISOR task will reset its task monitoring variable, activate the USER task and terminate itself. The activated USER task driver will check its subroutine monitoring variables and find that none of its subroutines have to be executed. Then it will check the task monitoring variable for the SUPERVISOR task, find the variable to be reset and reactivate all the other tasks. All the reactivated tasks will then terminate themselves since the SUPERVISOR task has completed its execution. The USER task driver which is a subroutine will return to the user main program and complete normally.

The implemented system was initially tested by running several simple examples taken directly from the GASPIV text book [PRIT74]. The program outputs were found to be identical to the results presented in the GASPIV text book. Other features of GASPIV such as monitor traces were also successfully tested on the multitasking system.

5.0 SUMMARY

The multitasking of system simulation is running successfully and is being used to emulate the parallel processor simulation system under development at Texas A & M University. The implemented system fully satisfies both the usage objectives and our design goals. Neither the existing language structure nor the execution structure of the GASPIV simulation language has been modified. The system is fully user transparent such that the users are not required to understand multitasking implementation. Also, the users have to no additional work to do other than the conventional GASPIV programming.

The important result achieved in this multitasking simulation system is the successful emulation of the architecture of the parallel processing

simulation system through the multitasking simulation system. This emulation of the parallel processor architecture has enabled the testing and the validation of the software architecture that will be implemented in the parallel processing simulation system. Another result achieved is the increase in memory made available for the simulation language on the TI 990/12 due to multitasking. Initially, the GASPIV package was implemented as a single task on the TI 990/12 and only very small problems (QSET of 1000) could be executed due to the 64K memory restriction for a single task on the system. The multitasking implementation has alleviated this restriction by the allocation of 64K for each individual task in the multitasking system and as a consequence has increased the problem size that could be executed to three fold compared to those executed under a single task implementation.

The next step in this research effort will be the implementation of the multitasking simulation system with appropriate modifications on the asynchronous parallel processing system which consists of off-the-shelf microcomputers so as to achieve main frame speed at low cost.

REFERENCES

[ANSI83] American National Standards Institute Inc., "Ada Programming Language," Department of Defense, 22 Jan 1983 ANSI/MIL-STD-1815A

[HOLT78] Holt, R.C., Graham, G.S., and et al, "Structured CONCURRENT PROGRAMMING with operating system Applications," Addison-Wesley Publishing Company, 1978

[JONE80] Jones, Anita K., and Schwarz, Peter, "Experience Using Multiprocessor Systems -- A Status Report," Computing Surveys Vol. 12, No. 2, June 1980 Pg. 121-165

[McLE83] McLean, Charles., Mitchell, Mary and Barkmeyer, Edward., "A Computer architecture for small-batch manufacturing ," IEEE Spectrum, May 1983 Pg. 59-64

[PEGD83] Pegden, C.Dennis., "Introduction to SIMAN," Systems Modeling Corporation State College, Pennsylvania, 1982

[PRIT74] Pritsker, A. Alan B., "The GASPIV Simulation Language," John Wiley & Sons New York, NY, 1974

[PRIT79] Pritsker, A. Alan B., and Pegden, C. D., "Introduction to Simulation and SLAM," Halsted Press, New York, NY 1979

[SMIT81] Smith, John M., Bernstein, Philip A., and et. al., "Multibase - integrating heterogenous distributed database systems," AFIPS - National Computer Conference Proceedings 1981, Volume 50 AFIPS Press. Arlington, Va 22209

[TIFT82] Texas Instruments 990 Model Computer "FORTRAN-78," Part No. 2268681-9701* C 1 Oct 1982

[TIFT83] Texas Instruments 990 Model Computer "FORTRAN-78 Programmer's Guide," Part No. 2268679-9701* C, 15 Dec 1983

[TIOS82] Texas Instruments 990 Model Computer "DX 10 Operating System Concepts and Facilities," Volumes I and II, Part No. 946250-9701* D, 1 Sep 1982

[TISV82] Texas Instruments 990 Model Computer "DX 10 Operating System Applications Programming Guide," Volume III, Part No. 0942650-9703* D, 1 Sep 1982

[YOUN84] Young, Robert E., and Krishnamurthi, Murali., "A Parallel Processor for System Simulation: The Design Rationale and Simulation Language Characteristics Suitable for Parallel Processing," Presented at the Spring IIE Conference on May 7th 1984, Chicago