ORNL/TM-9493

Dist. Category UC 20

Computing and Telecommunications Division

AN INTRODUCTION TO PROGRAMMING MULTIPLE-PROCESSOR COMPUTERS

H. R. Hicks and V. E. Lynch

ORNL/TM--9493

DE85 010274

DATE PUBLISHED – April 1985

NOTICE: This document contains information of a preliminary nature. It is subject to revision or correction and therefore does not represent a final report.

Prepared by the
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831
operated by
Martin Marietta Energy Systems, Inc.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-84OR21400

Conf 483 2600

ER

CONTENTS

ACKNOWLEDGMENTS

We are indebted to many people for our understanding of this subject. In particular, we thank Dieter Fuss, Tim Axelrod, Kirby Fong, Gary Long, Larry Diegel, Dave Storch, Bruce Curtis, Clark Streeter, Larry Berdahl, and their colleagues at the National Magnetic Fusion Energy Computer Center. We have benefited from the encouragement of Ben Carreras and our other colleagues at Oak Ridge, and especially from Lowell Charlton, whose MHD program provided us with a realistic environment in which to experiment. We are grateful to Jeff Holmes, Kirby Fong, and Mike Heath for critical readings of the manuscript. Finally, we thank Mike Heath, Eugene Miya, and T-C. Liu for assistance in locating related material

ABSTRACT

FORTRAN applications programs can be executed on multiprocessor computers in either a unitasking (traditional) or multitasking form. The latter allows a single job to use more than one processor simultaneously, with a consequent reduction in wall-clock time and, perhaps, the cost of the calculation. An introduction to programming in this environment is presented The concepts of synchronization and data sharing using EVENTS and LOCKS are illustrated with examples. The strategy of strong synchronization and the use of synchronization templates are proposed. We emphasize that incorrect multitasking programs can produce irreproducible results, which makes debugging more difficult

1. INTRODUCTION

Computers with multiple processors are increasingly available for large-scale scientific computation. This paper is intended to introduce FORTRAN programmers to this new environment. Since existing programs will continue to run on many multiprocessor computers (perhaps with some changes), we will first discuss the conditions under which it is desirable to modify them (or construct new ones) to take advantage of the multiple processors. For readers who decide to proceed, we discuss the two fundamental concepts of synchronization and data sharing. We believe that this introduction will provide sufficient background for most users, but those who want a definitive treatment will need to consult the bibliography.

This presentation will refer to the Cray X-MP or Cray-2 computers running under the Cray Timesharing System (CTSS). However many of the concepts are more widely applicable. We believe that the reader will find it easy to determine the applicability of any remark to his own situation. In particular, our analysis is appropriate for the class of computers with shared memory. This is in contrast to computers in which each processor has its own memory and data are transmitted from processor to processor as needed. Our discussion is also influenced by the fact that CTSS is a multiuser environment. Not only do users share memory, but different users may simultaneously use the processors. The strategies we suggest are not intended for strict adherence, but are designed to provide conceptual guidelines (from which it may be necessary to depart to an extent that depends on the applications problem as well as the computing environment).

A considerable volume of literature already exists on the subject of multiple-processor computers; however, we feel that there is a need to address the programming issues in terms familiar to applications programmers rather than to computer scientists. We have found that some presentations are overly complex. Moreover, some excellent articles are not generally available or cannot be referenced. Although synchronization and data sharing can be difficult to employ, they are not difficult concepts. The material presented here comes from our own experience, as well as constitutes a review of what we have learned from others. We treat the subject by way of a few simple examples. We hope the reader will be able to infer from these most of what is necessary to produce multitasking programs

We will regard each processor as an essentially complete computer. This means that the N-processor computer can execute N independent jobs as long as they all can fit into the common memory. We assume that such a mode of operation requires no effort on the part of the user and that the operating system will tend to the details. Traditional jobs which are executed with a single stream of instructions are called UNItasking jobs. The only way a program will use more than one processor at a time is if the programmer takes the necessary steps to organize his program into sections (called "tasks") that can be performed simultaneously. These tasks need not be totally independent, but the more independent they are, the easier the programming and the more efficient the results.

Just as high-level languages such as FORTRAN allow the user to ignore hardware details, multitasking in such languages can be accomplished at the same high level. In fact, even the number of processors available need not be known. Thus, we can approach multitasking at the FORTRAN level. The user, in general, will not control which processor executes each task, nor will the precise order of execution of instructions which are in separate tasks be predictable. Because of this, multitasking programs have a profound new property. A program that happens to be incorrect may produce different results on subsequent executions with the same input data. Thus, a correct result will no longer guarantee a correct code, even for the particular logic path tested. We shall recommend ways to minimize the probability of producing an incorrect code. Prevention is the key here because, in general, one does not know when an irreproducible code exists, and even if one does know, debugging runs are more difficult because they themselves are not necessarily reproducible.

A good strategy is to first structure the code so that it is suitable for multitasking. This part requires the greatest effort and care. It implies identifying the sections of the code which can be executed in parallel. Next, if necessary, the code should be sped up by conventional programming techniques, such as vectorization. Finally, when this has been tested, the multitasking capability should be added. Tasks can be used at any level of code logic where parallelism exists or can be exploited. However, since there is some cost to creating tasks, one should try to multitask at relatively high levels.

There is another matter which, strictly speaking, has nothing to do with multitasking, but which will affect many users converting from single-processor computers to multiprocessor computers: data initialization and retention. FORTRAN rules do not specify the contents of an uninitialized variable or the contents of a local variable on a subsequent execution of a subroutine. However, because many traditional loaders provide zero initial values and many compilers retain the values of local variables for use when the same subroutine is executed subsequently, many existing programs rely on these conditions. Multiprocessor computers generally assign space for local variables at the time a subroutine is called. It is likely that this space was previously used for another purpose. Thus, on multiprocessors, even unitasking jobs must generally adhere to the practice of assigning initial values to each local variable on each entry into a routine. When the value of a local variable must be retained from the previous subroutine call, this can be accomplished by specifying the variable in a FORTRAN `SAVE` statement or by putting it in a `COMMON` block.

2. POSSIBLE ADVANTAGES OF MULTITASKING

For the user, there are two primary advantages to multitasking reducing wall-clock time for execution and/or reducing the cost of the job. Multitasking will almost always reduce the wall-clock time. Of course, this is only important for long-running jobs that take greater than the desired turnaround time or too large a fraction of the mean time between failures of the machine. In a timesharing environment, multitasking could improve the response time as well.

Whether multitasking reduces cost depends on the charging algorithm. With an operating system that gives the machine to a single user, one would suppose that the charge is proportional to the total residency time. In this case, multitasking would generally reduce the charge However, with operating systems such as CTSS in which users can share both memory and processors, the advantage is substantially reduced. Even in this environment, however, if there is a substantial memory residency charge, multitasking may pay off.

The user needs to weigh these two potential gains against the effort of creating and maintaining a multitasking code Moreover, the structure necessary for an optimal multitasking code will compete with other design considerations, such as minimizing the number of operations or the memory size. In an environment such as a Cray X-MP or Cray 2 running under CTSS, we expect that multitasking will be worthwhile for only a small fraction of jobs, namely, those characterized by very large memory and/or very long run times.

3. TASK SYNCHRONIZATION

In this section, we illustrate how the user starts tasks and, once started, how the order of execution is controlled to reflect dependencies among the tasks. The burden falls entirely on the user to decide which calculations can be performed in parallel. The tasks should be organized to reflect the logic of the program and should generally not attempt to conform exactly to the number of processors. In multiuser environments, it is generally not advantageous to try to maintain a constant number of tasks.

Generally, a SUBROUTINE is the smallest FORTRAN unit that can be a task. In fact, creating a task is analogous to CALLing a SUBROUTINE, except that the CALLing routine continues to execute beyond the CALL and the CALLed routine never returns to the CALLer, but instead terminates.

When execution commences, only one task exists. We shall refer to this as the "original" task, even though, once additional tasks are created, they are in some sense equal. For many applications, it is conceptually easier to make the original task a controlling task and to allow the other tasks to perform specific bits of work. We recommend this conceptual approach as being safer, at least for inexperienced users. Therefore, we shall assume that the original task will do the job of starting all other tasks. If the work of the other tasks is similar, the top-level routine in each might be the same, but this is not necessary.

Consider a unitasking program containing

```
C     EXAMPLE 1A
      CALL RED(R1,R2, ...)
      CALL GREEN(G1,G2, ...)
      CALL BLUE(B1,B2, ...)
```

If the three calculations are independent, then instead of the three calls, three tasks could be started:

```
C     EXAMPLE 1B (INCOMPLETE)
      EXTERNAL RED,GREEN,BLUE
      CALL TSKSTART(TCA(1,1),RED,R1,R2, ...)
      CALL TSKSTART(TCA(1,2),GREEN,G1,G2, ...)
      CALL TSKSTART(TCA(1,3),BLUE,B1,B2, ...)
```

One should think of this as creating a situation in which as many as four tasks (including the original task) are executing after these statements are executed. In actuality, some tasks may be completed before others have started. As long as the unitasking version is correct and the tasks RED, GREEN, BLUE, and the original task are independent, the multitasking version is correct. Some additional declarations are necessary to make this example complete; they are described later, along with the TCA array. We use the CTSS syntax here, but some other implementations have similar capabilities. For example, on the Denelcor HEP a task is started with

```
      CREATE RED(R1,R2, ...)
```

Suppose now that the original task should proceed only after the completions of RED. GREEN. and BLUE This is ensured by writing

```
C     EXAMPLE 1C
      INTEGER TCAR,TCAG,TCAB
      COMMON/TNAME/TCAR(2),TCAG(2),TCAB(2)
      EXTERNAL RED,GREEN,BLUE
      TCAR(1)=2
      TCAG(1)=2
      TCAB(1)=2
      CALL TSKSTART(TCAR,RED,R1,R2,     )
      CALL TSKSTART(TCAG,GREEN,G1,G2,     )
      CALL TSKSTART(TCAB,BLUE,B1,B2, ...)
      CALL TSKWAIT(TCAB)
      CALL TSKWAIT(TCAG)
      CALL TSKWAIT(TCAR)
C     RED, GREEN, AND BLUE HAVE COMPLETED
```

In the CTSS implementation of multitasking. the integer task control arrays (here TCAR. TCAG. and TCAB) are objects. consisting of two or more elements. that are associated with each task. In this case they associate each TSKWAIT with an appropriate TSKSTART. and each TCA is two elements. The user must store the number of elements in the first element: hence TCAR(1)=2. A TCA with more elements allows the user to pass more information into the task. but we shall not illustrate that here. In example 1C. the original task will wait for each of the other tasks to be completed in turn before proceeding. The order of the TSKWAIT's is immaterial in this example. The TSKSTART and TSKWAIT for BLUE could be replaced with a CALL BLUE so that the original task. which would otherwise be waiting anyway. would do this work. However. one should not be overly concerned about tasks left waiting. Choices such as this should be resolved on the basis of creating the most understandable code. rather than on "optimization."

Generally. the gain of multitasking is greatest if the execution times of RED. GREEN. and BLUE are equal. However. it is usually not worthwhile to worry about this unless one suspects that the execution times are greatly disparate. in which case the code approaches the efficiency of a unitasking code plus the multitasking overhead. Even if the execution times are disparate. if the number of tasks is greater than the number of processors. multitasking may be rewarded since short tasks may execute serially on one processor while long ones execute on other processors. In this sense. the operating system may provide a form of dynamic load leveling.

Another common situation in which multitasking may apply is

```
C     EXAMPLE 2A
      DO 10 N=1,NNAX
10    CALL WORK(N,W1,W2, ...)

      END
      SUBROUTINE WORK(NN,W1,W2,...)
      COMMON A(100,100),B(100),C(100)
      DO 10 J=1,100
10    A(NN,J)=NN*B(J)+C(J)
      RETURN
      END
```

Usually one would prefer that the subroutine to be multitasked contain more work, but the example is intended to illustrate the case in which the NMAX executions of WORK can be performed in parallel:

```
C     EXAMPLE 2B
      INTEGER TCA
      COMMON/TNAME/TCA(2,100)
      DIMENSION NARRAY(100)
      EXTERNAL WORK
      DO 10 N=1,NMAX
      TCA(1,N)=2
      NARRAY(N)=N
10    CALL TSKSTART(TCA(1,N),WORK,NARRAY(N),W1,W2,...)
      DO 20 N=1,NMAX
20    CALL TSKWAIT(TCA(1,N))
```

Thus, the original task will start the NMAX copies of WORK and wait for them all to be completed. Subroutine WORK is not altered. Note that it would have been incorrect to pass N as the argument, since the value of variable N continues to change after each task is started. This crucial point will be addressed further in the next section. Any task can start new tasks at any time as long as the necessary synchronization is provided for all of the tasks that can be running at any time.

Now consider a more complicated unitasking code:

```
C     EXAMPLE 3A
      DO 10 N=1,NMAX
10    CALL WORK1(N,W11,W12,...)
```

```
C     COULD DO WORK HERE IN ORIGINAL TASK
      DO 20 N=1,NMAX
20    CALL WORK2(N,W21,W22,    )
      .
      .
      END
```

The comment line "COULD DO WORK HERE..." indicates the placement of work which must follow the completion of all executions of WORK1 and precede all executions of WORK2. Suppose that WORK1 and WORK2 can each be multitasked, but all of the WORK1's must be completed before any of the WORK2's start. The safest approach is to create additional tasks

```
C     EXAMPLE 3B
      INTEGER TCA
      COMMON/TNAME/TCA(2,200)
      DIMENSION NARRAY(100)
      EXTERNAL WORK1,WORK2
      DO 10 N=1,NMAX
      TCA(1,N)=2
      NARRAY(N)=N
10    CALL TSKSTART(TCA(1,N),WORK1,NARRAY(N),W11,W12,...)
      DO 15 N=1,NMAX
15    CALL TSKWAIT(TCA(1,N))
C     COULD DO WORK HERE IN ORIGINAL TASK
      DO 20 N=1,NMAX
      TCA(1,NMAX+N)=2
20    CALL TSKSTART(TCA(1,NMAX+N),WORK2,NARRAY(N),W21,W22,...)
      DO 25 N=1,NMAX
25    CALL TSKWAIT(TCA(1,NMAX+N))
      .
      .
      END
```

In principle, this technique of totally independent tasks is sufficient. However, consider the addition of an overall DO loop around example 3B. This could result in a very large number of TSKSTART's being executed. There is an overhead cost associated with starting new tasks, so execution efficiency may suffer. To reduce the necessary number of TSKSTART's and increase the amount of calculation in each task, some tools have been developed which allow tasks to cooperate during their execution. In this example we could combine WORK1 and WORK2 into a single task for each N. One could use EVENTS to synchronize the NMAX tasks as each one finishes

WORK1. An event is like a bit (with two states, POSTED and CLEARED) that all tasks can see. The user can create as many events as he desires. There are three operations that a task can perform with respect to each event: post, wait, and clear. Generally, one task will alternately post and clear a given event and certain other tasks will wait for the event to be posted. In example 2B, the TSKWAIT in the original task could have been replaced with event waits,

```
      DO 20 N=1,NMAX
20    CALL EVWAIT(ET(N))
```

and at the end of subroutine WORK, one would have

```
      CALL EVPOST(ET(NN))
```

As each WORK task is completed, it posts the ET event that corresponds to that task. When all NMAX events have been posted, the original task breaks out of the EVWAIT loop.

Clearly, this example can be generalized to establish a number of synchronization points within tasks. Synchronization points can involve all or just some tasks. For synchronization above the simplest level, extreme care should be exercised because an incorrect event structure may not be apparent in the results of the calculation. We recommend two ways to minimize this danger: STRONG SYNCHRONIZATION and use of synchronization templates.

Strong synchronization refers to the practice of making the event structure robust by using more than the minimum number of events. This will also reduce the chance that subsequent program changes will introduce an error into a correct event structure. The following example has this property, as well as providing a tested synchronization template. We will use 2*(NMAX+1) events.

Consider a unitasking code that generalizes example 3A with the addition of an overall loop:

```
C     EXAMPLE 4A
      :
      DO 100 J=1,JMAX
C     COULD DO WORK HERE IN ORIGINAL TASK
      DO 10 N=1,NMAX
10    CALL WORK1(N,W11,W12, ...)
C     COULD DO WORK HERE IN ORIGINAL TASK
      DO 20 N=1,NMAX
20    CALL WORK2(N,W21,W22, ...)
100   CONTINUE
      :
      END
```

This could correspond, for example, to a time-stepping or iteration loop in which each step or iteration calls for two blocks of work, each of which can be multitasked

One way to visualize the implementation of example 4 with events is to show the time sequence within the original and other tasks and the points (indicated by arrows) at which control is passed between the original task and the other tasks:

```
         ORIGINAL TASK                         WORK(N)
           start WORK (N)
           DO 100
           wait for all ET(1,N)    ←     30    post ET(1,N)
           do work
           clear EN(2)
           post EN(1)              →           wait for EN(1)
                                               do work
                                               clear ET(1,N)
           wait for all ET(2,N)    ←           post ET(2,N)
           do work
           clear EN(1)
           post EN(2)              →           wait for EN(2)
    100    CONTINUE                            do work
                                               clear ET(2,N)
           wait for all ET(1,N)                GO TO 30
```

The original task starts NMAX copies of WORK and then waits until all NMAX events ET(1;N) have been posted; then before posting event EN(1) it can perform some work while the other tasks are idle. The posting of EN(1) by the original task is the signal that all the other tasks can commence their first block of work. Note that events are not cleared just after being posted, because that provides no guarantee that the tasks looking for the event will see it while it is posted. In this template, events are always cleared after waiting for a different event which confirms that the original event was seen. The sequence *post-wait-work-clear* can be repeated any number of times. In each such block, *post* and *clear* refer to the same event and *wait* refers to an event posted by another task or tasks. The *work-clear* order can be reversed as long as *work* and *clear* are sandwiched in between the *wait* and the following *post*. The expression of this event structure in CTSS FORTRAN is

```
C     EXAMPLE 4B
      INTEGER TCA,ET,EN
      COMMON/TNAME/TCA(2,100)
      COMMON/EVCON/ET(2,100),EN(2)
      DIMENSION NARRAY(100)
      EXTERNAL WORK
      .
      CALL EVASGN(EM(1),ASTAT)
      CALL EVASGN(EM(2),ASTAT)
      DO 10 N=1,NMAX
      TCA(1,N)=2
      NARRAY(N)=N
      CALL EVASGN(ET(1,N),ASTAT)
      CALL EVASGN(ET(2,N),ASTAT)
10    CALL TSKSTART(TCA(1,N),WORK,NARRAY(N),W11,W21,W12,W22, ...)
      DO 100 J=1,JMAX
      DO 11 N=1,NMAX
11    CALL EVWAIT(ET(1,N))
C     COULD DO WORK HERE (OTHER TASKS IDLE)
      CALL EVCLEAR(EM(2))
      CALL EVPOST(EM(1))
      DO 12 N=1,NMAX
12    CALL EVWAIT(ET(2,N))
C     COULD DO WORK HERE (OTHER TASKS IDLE)
      CALL EVCLEAR(EM(1))
      CALL EVPOST(EM(2))
100   CONTINUE
      DO 20 N=1,NMAX
20    CALL EVWAIT(ET(1,N))
      :
      END
      SUBROUTINE WORK(NN,W11,W21,W12,W22, ...)
      INTEGER ET,EM
      COMMON/EVCOM/ET(2,100),EM(2)
30    CALL EVPOST(ET(1,NN))
      CALL EVWAIT(EM(1))
C     ORIGINAL TASK IDLE
      CALL WORK1(NN,W11,W12, ...)
      CALL EVCLEAR(ET(1,NN))
      CALL EVPOST(ET(2,NN))
```

```
              CALL EVWAIT(EN(2))
        C     ORIGINAL TASK IDLE
              CALL WORK2(NN,W21,W22, ...)
              CALL EVCLEAR(ET(2,NN))
              GO TO 30
              END
```

Note that after the first trip through statement 10, more than one task is active. The comments indicate the points in each task at which synchronization allows work to be done.

In SUBROUTINE WORK we have chosen to CALL WORK1 and WORK2 to isolate the event structure from the rest of the code. If large sections of in-line FORTRAN appeared here, it is possible that an (erroneous) GO TO could go from WORK1 to WORK2, skipping a clear-post-wait sequence. However, the event structure has sufficiently strong synchronization to detect this "break" in the event structure. The code will run to a deadlock. It is always preferable to deadlock than to have the code proceed with the tasks potentially out of step.

Another way to visualize this event structure is to show the time sequence organized by event. For each event, we show the posting, the clearing, and the waiting. [When the original task initiates the action, upper case (P,C,W) is used; when it refers to the WORK tasks, lower case (p,c,w) is used.]

ET(1,N)	EN(1)	ET(2,N)	EN(2)
W,p			
			C
	w,P		
c			
		W,p	
	C		
			w,P
		c	

Work can be done in the main task just before or after C, and work can be done in the other tasks just before or after c. Note that in the FORTRAN for example 4B, there is a final wait for ET(1,N) to ensure that the WORK tasks are all idle before proceeding. To generalize this example to perform more blocks of work within each task, simply extend the pattern. Each additional block adds two columns and two rows to the pattern.

Although events are powerful, they are not convenient for some situations. Consider the case of a variable that must be modified by each task in any order. It is not necessary to synchronize the tasks all at once; one must merely prevent the simultaneous modification of the variable by more than one task. This can easily be accomplished with a "lock" which, like an event, is an object with two possible states. The statement

```
CALL LOCKON(LOCKNAME)
```

causes the lock to be set if it is not and causes the task to wait for the lock to be turned off if it is already set. This contrasts with events. Posting an event that is already posted has no effect. The statement

```
CALL LOCKOFF(LOCKNAME)
```

clears a lock and continues. As with events, locks must be assigned

```
CALL LOCKASGN(LOCKNAME)
```

before they are used. Before each task modifies the shared variable(s), it locks the lock; afterward it turns the lock off:

```
CALL LOCKASGN(LOCKA)
  ⋮
CALL LOCKON(LOCKA)
A=A+1
CALL LOCKOFF(LOCKA)
```

The effect of this, assuming that each task contains similar coding, is that the statement in which shared variables are modified (here A=A+1) is not executed simultaneously by different tasks. In contrast to our examples for events, this does not cause all the tasks to wait for a common signal (an event); it merely prohibits the simultaneous execution of the critical section of code — but the order in which the tasks execute the critical section is unpredictable.

4. DATA SHARING

Depending on how they are declared, variables are visible to one or more tasks. Variables that are visible to more than one task must be treated carefully to ensure that they are used and assigned in the proper sequence by different tasks. The seriousness of this issue is emphasized by considering an apparently isolated section of code. If that section of code can be executing simultaneously with another task, events and/or locks must generally be used when referring to shared data.

In traditional FORTRAN, variables can be classed as local, COMMON, or dummy argument. The scope of a variable becomes more complex in a multitasking code, as shown in the table

Unitasking	Multitasking	
	Shared	Not Shared
LOCAL	SAVEd	LOCAL
COMMON	COMMON	TASK COMMON
DUMMY ARGUMENT	depends	

The CTSS generalization of COMMON is not defined at the time of this writing, so we use the nomenclature established by Cray Research, Inc. Local variables that are not SAVEd are only defined during the execution of the routine in which they appear. Their value is not retained after RETURN from the routine. For SUBROUTINEs in more than one task, local variables are not shared between tasks. Separate copies of each local variable exist for each task. When local variables are SAVEd, two things happen, both being a consequence of the fact that SAVEd variables are assigned to a single static location. The same variable value is available to all tasks. The value is retained for subsequent executions of the subroutine in the same or another task. During a period of time in which a SAVEd variable is not modified by any task, it can be used by all tasks as a constant. Any modification of SAVEd variables should be controlled by the use of events and locks. Clearly, each task could be allowed to modify independent elements of a SAVEd array without synchronization.

All four possibilities of sharing or not sharing between routines and tasks are available:

	Shared Between	
	Routines	Tasks
local	no	no
SAVEd	no	yes
TASK COMMON	yes	no
COMMON	yes	yes

Variables in COMMON are global with respect to both tasks and routines. A new declaration syntax,

```
TASK COMMON/CNAME/V1,V2, ...
```

creates a COMMON having a separate copy for each task. Thus a variable in a TASK COMMON (as with a local variable) can simultaneously have different values in each task.

It is a deeply ingrained notion in traditional programming that within a segment of in-line code all modifications of the values of variables are apparent. The occurrence of SUBROUTINE or FUNCTION invocations can result in modification of arguments and variables in COMMON. In a multitasking code, shared variables can be changing unpredictably during the course of execution, so it is essential to have a clear mental picture of which variables are shared among tasks.

The scope of dummy SUBROUTINE arguments depends first on the declaration of the actual (original) argument. A variable that is originally local with respect to tasks will be shared among tasks if it is passed in a TSKSTART invocation. However, we recommend caution when passing arguments into tasks, because this creates numerous potential failure modes. If the CALLing routine terminates before the task that it started terminates, not only do variables local in the CALLing routine become undefined, but, in some implementations, the addresses of all arguments passed to a task may also become unreliable. It is for this reason, in example 4B above, that we use COMMON to communicate the events to the tasks. In examples 3B and 4B, the task index N is passed into the tasks as an element of the array NARRAY(N)=N. To pass N itself from the original task would result in all tasks referencing the same location N, rather than obtaining the value of N present at the time of the TSKSTART.

It is also necessary to modify the notion of arguments preserved on exit from a SUBROUTINE. Consider

```
CALL LIN(B,A,C)
```

and assume that the arguments A and C are input arguments; that is, they are unchanged on exit from LIN. Suppose that the multitasked subroutine WORK in example 2B consists of

```
C     EXAMPLE 5A
      SUBROUTINE WORK(NN)
      COMMON A,B(100),C(100)
      CALL LIN(B,A,C,NN)
      RETURN
      END
      SUBROUTINE LIN(B,A,C,NNN)
      DIMENSION B(100),C(100)
      B(NNN)=B(NNN)+C(NNN)*A
      RETURN
      END
```

Variables A, B, and C are shared, but the simultaneous calls to LIN are correct because within the tasks, A and C are unchanged, so all tasks can use them. Variable B is modified, but each task can only modify one element. However, a modification to LIN that retains the property that A and C are unchanged on exit from LIN results in an incorrect task:

```
C     EXAMPLE 5B(INCORRECT)
      SUBROUTINE WORK(NN)
      COMMON A,B(100),C(100)
      CALL LIN(B,A,C,NN)
      RETURN
      END
      SUBROUTINE LIN(B,A,C,NNN)
      DIMENSION B(100),C(100)
      A=A*C(NNN)
      B(NNN)=B(NNN)+A
      A=A/C(NNN)
      RETURN
      END
```

This was intended to produce the same results as example 5A. The problem is that one copy of variable A is shared among all the tasks. Within each task, A is temporarily multiplied by the element of C associated with that task. It is quite possible that another task will pick up A before it is restored to its original value. Thus, the results of example 5B will be irreproducible unless locks or events are employed. Unfortunately, such an incorrect code may execute numerous test runs correctly, failing to reveal its lack of correctness.

5. SUMMARY

Multitasking FORTRAN programs are more susceptible to error and considerably more difficult to debug than unitasking codes. The results of an incorrect multitasking code may be irreproducible, and thus may be correct in any given run. The following guidelines provide a starting point optimized for CTSS. They will not be equally true in other environments.

To multitask or not? Multiprocessor computers may run unitasking (traditional) programs. Multitasking can reduce wall-clock time, computer charge, and response time. Multitasking adds a system overhead cost in addition to being in competition with other code design objectives such as readability, minimum memory, etc. One should not multitask a code without the clear prospect of a net gain.

Strategy: (1) organize the program so that it will be suitable for multitasking; (2) employ conventional optimization (such as vectorization) from the bottom up; and (3) multitask from the top down. Tasks should contain enough work to overcome the overhead of starting them. Multitask at a level (or levels) at which the program has natural parallelism. Do not try to match the number of tasks to the number of processors—match the number of tasks to the problem. In a multiuser environment, it is not worthwhile to try to maintain a constant number of tasks.

Task synchronization and data sharing should be planned carefully to obtain a correct, efficient code. It is safest to use an existing, tested synchronization template (event and/or lock structure). Use STRONG SYNCHRONIZATION—an overdesigned synchronization scheme with more than the minimum number of events and/or locks. This will help prevent tasks from getting out of step. Strong synchronization is designed to increase the probability that an incorrect program will go to an error condition or deadlock.

SAVEd and COMMON variables and some dummy arguments are visible to, and can be modified by, different tasks. Use documentation and programming conventions (e.g., naming conventions) to make such shared variables apparent. Minimize passing arguments into tasks. Access to shared variables must be controlled with events and/or locks.

BIBLIOGRAPHY

1. G. Andrews and F. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys* **15** (1983), 3.

2. J. McGraw and T. S. Axelrod, "Exploiting Multiprocessors: Issues and Options," LLNL Report UCRL-91734, October 31, 1984.

3. J. L. Baer, "A Survey of Some Theoretical Aspects of Multiprocessing," *ACM Computing Surveys* **5** (1973), 31.

4. G. J. Blair, "Reentrancy and Multitasking on Cray Computers," LLNL Report UCID-30199, May 15, 1984.

5. P. Brinch Hanzen, "Concurrent Programming Concepts," *ACM Computing Surveys* **5** (1973), 223.

6. Cray Research, Inc., "Multitasking User's Guide," Cray Computer Systems Technical Note SN-0222 (February 1984), Mendota Heights, Minnesota.

7. K. Fong, "Multitasking," *NMFECC Buffer* 8(9), 2 (September 1984).

8. K. Fong, "Locks and Events in Multitasking," *NMFECC Buffer* 8(10) (October 1984).

9. IEEE, Proceedings of the International Conferences on Parallel Processing, Columbus, Ohio, Aug. 26-29, 1980; Bellaire, Michigan, Aug. 24-27, 1982 and Aug. 23-26, 1983.

10. Proceedings of the Fourth Summer School on Computational Physics, Stara Lesna, Czechoslovakia, May 19-28, 1981, in *Comput. Phys. Commun.* **26** (1982), 237.

11. National Magnetic Fusion Energy Computer Center, "MPDOC" (on-line documentation).

12. E. Lusk and R. Overbeek, "Implementation of Monitors with Macros: a Programming Aid for the HEP and Other Parallel Processors," ANL-83-97, Argonne National Laboratory, December 1983.

13. M. Ben-Ari, *Principles of Concurrent Programming*, Prentice-Hall, 1982.

14. P. Brinch Hanzen, *The Architecture of Concurrent Programs*, Prentice-Hall, 1977.

15. R. C. Holt, et al., *Structured Concurrent Programming*, Addison-Wesley, 1978.

ORNL/TM-9493
Dist. Category UC-20

INTERNAL DISTRIBUTION

1. B. A. Carreras
2. W. A. Cooper
3. E. C. Crume, Jr.
4. L. Garcia
5. M. T. Heath
6. J. A. Rome
7. S. E. Attenberger
8. J. L. Cantrell
9. L. A. Charlton
10. D. N. Clark
11. R. H. Fowler
12. J. D. Galambos
13. J. A. Holmes
14. D. K. Lee
15. R. W. McGaffey
16. R. N. Morris
17. J. K. Munro
18. C. W. Nestor, Jr.
19. L. W. Owen
20. K. E. Rothe
21. D. J. Strickler
22. J. S. Tolliver
23. T. C. Tucker
24. W. I. van Rij
25. G. E. Whitesides

26–40. H. R. Hicks
41–55. V. E. Lynch
56–57. Laboratory Records Department
58. Laboratory Records, ORNL-RC
59. Document Reference Section
60. Central Research Library
61. Fusion Energy Division Library
62–63. Fusion Energy Division Publications Office
64. ORNL Patent Office

EXTERNAL DISTRIBUTION

65. D. V. Anderson, Lawrence Livermore National Laboratory, P.O. Box 5509, Livermore, CA 94550
66. P. Andrews, GA Technologies, Inc., P.O. Box 81608, San Diego, CA 92138
67. A. Y. Aydemir, Institute for Fusion Studies, University of Texas, Austin, TX 78712
68. M. Azumi, Japan Atomic Energy Research Institute, Tokai-mura, Naka-gun, Ibaraki-ken, Japan
69. D. C. Barnes, Institute for Fusion Studies, University of Texas, Austin, TX 78712
70. R. H. Berman, Massachusetts Institute of Technology, 77 Massachusetts Ave., Cambridge, MA 02139
71. L. C. Bernard, GA Technologies, Inc., P.O. Box 81608, San Diego, CA 92138
72. O. Buneman, ERL Stanford, Stanford, CA 94305
73. E. J. Caramana, Los Alamos National Laboratory, Ctr. 6, MS 642, Los Alamos, NM 87545
74. M. S. Chu, GA Technologies, Inc., TO 500, P.O. Box 81608, San Diego, CA 92138
75. B. I. Cohen, L630, Lawrence Livermore National Laboratory, P.O. Box 5511, Livermore, CA 94550
76. J. Delucia, Plasma Physics Laboratory, Princeton University, P.O. Box 451, Princeton, NJ 08544
77. F. J. Helton, Fusion Division, GA Technologies, Inc., Box 81608, San Diego, CA 92138
78. D. W. Hewett, Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94550
79. S. C. Jardin, Plasmas Physics Laboratory, Princeton University, P.O. Box 451, Princeton, NJ 08544

80. C. Karney, Plasma Physics Laboratory, Princeton University, P.O. Box 451, Princeton, NJ 08544
81. G. D. Kerbel, Lawrence Livermore National Laboratory, P.O. Box 5509, Livermore, CA 94550
82. G-I. Kurita, Japan Atomic Energy Research Institute, Tokai-mura, Naka-gun, Ibaraki-ken, Japan
83. L. L. Lao, GA Technologies, Inc., P.O. Box 81608, San Diego, CA 92138
84. J. K. Lee, GA Technologies, Inc., P.O. Box 81608, San Diego, CA 92138
85. C. C. Lilliequist, Los Alamos National Laboratory, P.O. Box 1663, Los Alamos, NM 87545
86. J. Manickam, Plasma Physics Laboratory, Princeton University, P.O. Box 451, Princeton, NJ 08544
87. F. W. McClain, GA Technologies, Inc., P.O. Box 81608, San Diego, CA 92138
88. B. McNamara, Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94550
89. A. A. Mirin, Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94550
90. D. A. Monticello, Plasma Physics Laboratory, Princeton University, P.O. Box 451, Princeton, NJ 08544
91. W. Park, Plasma Physics Laboratory, Princeton University, P.O. Box 451, Princeton, NJ 08544
92. G. Rewoldt, Plasma Physics Laboratory, Princeton University, P.O. Box 451, Princeton, NJ 08544
93. D. D. Schnack, Science Applications, Inc., La Jolla, CA 92038
94. A. G. Sgro, Los Alamos National Laboratory, P.O. Box 1663, MS 642, Los Alamos, NM 87545
95. A. I. Shestakov, L-561, Lawrence Livermore National Laboratory, P.O. Box 5509, Livermore, CA 94550
96. T. Takeda, Japan Atomic Energy Research Institute, Tokai-mura, Naka-gun, Ibaraki-ken, Japan
97. Y. Tanaka, Fujitsu Limited, 1-17-25, Shinkamata, Ohta, Tokyo 144, Japan
98. R. E. Waltz, Fusion Dept. TO-521, GA Technologies, Inc., P.O. Box 81608, San Diego, CA 92138
99. R. M. Wieland, Plasma Physics Laboratory, Princeton University, P.O. Box 451, Princeton, NJ 08544
100. Office of the Assistant Manager for Energy Research and Development, Department of Energy, Oak Ridge Operations, Box E, Oak Ridge, TN 37830
101. J. D. Callen, Department of Nuclear Engineering, University of Wisconsin, Madison, WI 53706
102. R. W. Conn, Department of Chemical, Nuclear, and Thermal Engineering, University of California, Los Angeles, CA 90024
103. S. O. Dean, Director, Fusion Energy Development, Science Applications, Inc., 2 Professional Drive, Gaithersburg, MD 20760
104. H. K. Forsen, Bechtel Group, Inc., Research Engineering, P.O. Box 3965, San Francisco, CA 94105
105. J. R. Gilleland, GA Technologies, Inc., Fusion and Advanced Technology, P.O. Box 85608, San Diego, CA 92138
106. R. W. Gould, Department of Applied Physics, California Institute of Technology, Pasadena, CA 91125
107. R. A. Gross, Plasma Research Laboratory, Columbia University, New York, NY 10027
108. D. M. Meade, Plasma Physics Laboratory, Princeton University, P.O. Box 451, Princeton, NJ 08544
109. P. J. Reardon, Princeton Plasma Physics Laboratory, P.O. Box 451, Princeton, NJ 08544

110. W. M. Stacey, Jr., School of Nuclear Engineering, Georgia Institute of Technology, Atlanta, GA 30332
111. G. A. Eliseev, I. V. Kurchatov Institute of Atomic Energy, P.O. Box 3402, 123182 Moscow, U.S.S.R.
112. V. A. Glukhikh, Scientific-Research Institute of Electro-Physical Apparatus, 188631 Leningrad, U.S.S.R.
113. I. Shpigel, Institute of General Physics, Academy of Sciences, Ulitsa Vauilova, 38, Moscow, U.S.S.R.
114. D. D. Ryutov, Institute of Nuclear Physics, Siberian Branch of the Academy of Sciences of the U.S.S.R., Sovetskaya St. 5, 630090 Novosibirsk, U.S.S.R.
115. V. T. Tolok, Kharkov Physical-Technical Institute, Academical St. 1, 310108 Kharkov, U.S.S.R.
116. R. Varma, Physical Research Laboratory, Navrangpura, Ahmedabad, India
117. Bibliothek, Max-Planck Institut fur Plasmaphysik, D-8046 Garching bei Munchen, Federal Republic of Germany
118. Bibliothek, Institut fur Plasmaphysik, KFA, Postfach 1913, D-5170 Julich, Federal Republic of Germany
119. Bibliotheque, Centre de Recherches en Physique des Plasmas, 21 Avenue des Bains, 1007 Lausanne, Switzerland
120. Bibliotheque, Service du Confinement des Plasmas, CEA, B.P. 6, 92 Fontenay-aux-Roses (Seine), France
121. Documentation S.I.G.N., Departement de la Physique du Plasma et de la Fusion Controlee, Centre d'Etudes Nucleaires, B.P. No. 85, Centre du Tri, 38041 Cedex, Grenoble, France
122. Library, Culham Laboratory, UKAEA, Abingdon, Oxfordshire, OX14 3DB, England
123. Library, FOM Instituut voor Plasma-Fysica, Rijnhuizen, Jutphaas, The Netherlands
124. Library, Institute of Physics, Academia Sinica, Beijing, Peoples Republic of China
125. Library, Institute of Plasma Physics, Nagoya University, Nagoya 64, Japan
126. Library, International Centre for Theoretical Physics, Trieste, Italy
127. Library, Laboratorio Gas Ionizzati, Frascati, Italy
128. Library, Plasma Physics Laboratory, Kyoto University, Gokasho Uji, Kyoto, Japan
129. Plasma Research Laboratory, Australian National University, P.O. Box 4, Canberra, A.C.T. 2000, Australia
130. Thermonuclear Library, Japan Atomic Energy Research Institute, Tokai, Naka, Ibaraki, Japan
131. D. Steiner, Rensselaer Polytechnic Institute, Nuclear Engineering Department, NES Building, Tibbets Avenue, Troy, NY 12181

132–237. Given distribution as shown in TID-4500, Magnetic Fusion Energy (Category Distribution UC-20)