JOURNAL OF COMPUTATIONAL PHYSICS 63, 140–156 (1986)

# An Introduction to Programming Multiple-Processor Computers*

H. R. HICKS AND V. E. LYNCH

*Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831*

Received February 13, 1985; revised April 29, 1985

FORTRAN applications programs can be executed on multipocessor 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 elapsed 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 irreducible results, which makes debugging more difficult. © 1986 Academic Press, Inc.

## 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

* Research sponsored by the Office of Fusion Energy, U.S. Department of Energy, under Contract DE-AC05-84OR21400 with Martin Marieta Energy Systems, Inc.

0021-9991/86 $3.00

(from which it may be necessary to depart depending 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 expect that 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 shared 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 or 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 does not 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 (i.e., at least on a subroutine level).

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 assomplished by specifying the variable in a FORTRAN SAVE statement, or by putting it in a COMMON block.

On the Denelcor HEP the questions of synchronization and data sharing are handled differently than described here. Variables which are shared between tasks have a state, either "empty" or "full," depending on whether the variable was last read from or written to memory, respectively. These access states can be used to control the interaction of tasks. Although there is no standard implementation of multitasking, all implementations we have seen provide equivalent functional capability.

## 2. Possible Advantages of Multitasking

For the user, there are two primary advantages to multitasking: reducing elapsed time for execution and/or reducing the cost of the job. Multitasking will almost always reduce the elapsed time. Of course, this is only important for long-running jobs that take greater than the desired turnaround time or a large 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. If the CPU charge is the sum of the idividual CPU charges, then this component of the cost is presumably not substantially affected, assuming that the total amount of calculation remains the same. However, if in addition there is a substantial memory charge, multitasking may pay off by reducing the memory residency time.

The user needs to weigh these potential gains against the effort of creating and maintaining a multitasking code. Moreover, the structure necessary for an optimal multitasking code will complete with other design considerations, such as minimizing the number of operations or the memory size. Once multitasking has been implemented in a code, all future modifications must take this into account. Moreover, the removal of multitasking from a code is not always an easy task. 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 what constitutes a task, 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 taks.

Generally, a SUBROUTINE is the smallest FORTRAN unit that can be a task. In fact, starting 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. More generally, each task contains several subrutines, one of which is initially invoked from another task and possibly others which are called from this "initial" one. A routine may appear in more than one task.

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. It is necessary to declare RED, GREEN, and BLUE to be EXTERNAL since these subroutine names are being passed as parameters. 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, NMAX
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, another approach is to allow tasks to synchronize with each other at various points in their execution. In some cases, this results in code which is easier to understand. For example, if in the unitasking program Example 3A, WORK1 and WORK2 would more naturally be coded as a single routine, such a structure could be maintained in a multitasking program by using synchronization. A secondary reason for using synchronization is based on execution efficiency. Consider the addition of an overall DO J = 1, JMAX loop around Example 3B. This could result in a very large

large number (2*NMAX*JMAX) of TSKSTART's being executed. There is an overhead cost associated with starting new tasks, so execution efficiency may suffer.

Some tools have been developed which allow tasks to cooperate during their execution. We illustrate their use by combining WORK1 and WORK2 into a single task ("WORK") for each N. We will show how EVENTS can be used 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. Posting an event allows all tasks waiting for that event to resume. Clearing an event stops the tasks at the next wait for that event until it is posted again. The user can create as many events as he desires. Just as for variables, the user chooses the names of events by declaring them. Arrays of events may also be used.

There are three operations that can be performed with respect to each event: post, wait, and clear. We will illustrate the case in which 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, one could add at the end of subroutine WORK

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

The purpose of this would be to indicate (to any other interested task) that its work is complete. Here, ET is an array of events which has been created by the user in a manner described later. Note that each of the NMAX tasks will post its own element of ET. In the original task, the TASKWAIT's would be replaced with

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

which will cause the original task to pause here until all of the other NMAX tasks have posted their corresponding events indicating completion. In this example, the effect is the same as the original form of Example 2B, but the use of EVENTS permits a convenient generalization to more complex situations. One penalty of this approach is that the modification to subroutine WORK makes it unsuitable (generally) for being invoked with a CALL.

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. The array of events ET will be posted and cleared in the NMAX subtasks and the array of events EN will be posted and cleared in the original task.

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) tasks
     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 (WORK1)
                                  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 (WORK2)
                                  clear ET(2, N)
     wait for all ET(1, N)        GO TO 30
```

The original task starts NMAX copies of WORK (which will combine WORK1 and WORK2) and then waits until all NMAX events ET(1, N) have been posted, indicating that the other tasks are ready to begin. This synchronization point accomplished nothing on the first trip through the DO 100 loop, but is needed on subsequent trips. Before posting event EN(1) the original task has the opportunity to 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 never cleared just after being posted, because that provides no guarantee that the tasks looking for the events will see them while they are 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/EVCOM/ET(2, 100), EN(2)
    DIMENSION NARRAY(100)
    EXTERNAL WORK
    :
    CALL EVASGN(EN(1), ASTAT)
    CALL EVASGN(EN(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(EN(2))
    CALL EVPOST(EN(1))
    DO 12 N = 1, NMAX
12  CALL EVWAIT(ET(2, N))
C   COULD DO WORK HERE (OTHER TASKS IDLE)
    CALL EVCLEAR(EN(1))
    CALL EVPOST(EN(2))
100 CONTINUE
    DO 20 N = 1, NMAX
20  CALL EVWAIT(ET(1, N))
    :
    END
```

```
      SUBROUTINE WORK(NN, W11, W21, W12, W22,...)
      INTEGER ET, EN
      COMMON/EVCOM/ET(2, 100), EN(2)
   30 CALL EVPOST(ET(1, NN))
      CALL EVWAIT(EN(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. Each event is created by a call to EVASGN, which returns an error flag as its second argument, as shown in Example 4B.

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. The number of events in Example 4B can be reduced to NMAX + 2, but we don't recommend it, because it then becomes more difficult to diagnose an error in the event structure.

Another way to visualize this event structure is to show the time sequence organized by event. For each event, we show the posting, clearing, and 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 statements 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 in unpredictable.

Events and locks may be used together. For example, the number of events in Example 4B could be reduced to about 3 by the addition of a few locks. It is generally true that events produce less parallelism, especially when all tasks but one are waiting for the last task to reach a synchronization barrier. However, the "looser," more execution-efficient structure that can result from using locks can also complicate the debugging process if the regions of code which can overlap are more extensive. Finally, the style of synchronization should strongly depend on what seems most natural for the given problem. This certainly includes the option of using unsynchronized tasks (only TSKSTARTs and TSKWAITs).

One way to obtain statistical evidence for the correctness of an event (or lock) structure is to program the structure, but with random amounts of busy work replacing any actual work within each task. Next, some code must be added to each task at the locations where work can occur, such that an overall calculation using shared variables can only yield a correct answer if the work is performed in the expected order. We tested Example 4B in this fashion. Between each synchronization point in the subtasks we incremented a counter corresponding to that task (in addition to the random busy work). At each point where the subtasks are supposed to be idle, the main task checks each counter for the correct value. If this procedure is executed a large number of times, and if sufficient CPU overlap is obtained, one can have some confidence in the events. This procedure assumes that the event structure is not data dependent, as would be the case if, for example, there were conditional jumps around synchronization points. The correctness of such a structure is even more difficult to determine.

## 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 | |

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 shared by all tasks which call the routine in which the variable is SAVEd. 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. When a SAVEd variable is passed as an actual argument to a routine, the corresponding dymmy argument is shared between tasks, since the scope is established by the original SAVE declaration.

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.

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, 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. Restoring the value of the shared variable A on exit from LIN is insufficient; A must remain unchanged in LIN. Unfortunately, incorrect code, such as Example 5B, may execute numerous test runs correctly, failing to reveal its lack of correctness.

It is a deeply ingrained notion in traditional programming that within a segment of code all modifications of the values of variables are apparent, with the exception of SUBROUTINE or FUNCTION invocations which can result in modification of arguments and any 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.

We have found that, even in a large code which uses events, it is feasible to employ a methodical technique to lcate errors in data sharing. First, each section of code which can overlap must be identified. Next, shared variables (i.e., COMMON, SAVEd, or some arguments) which are modified in these sections must be located. Finally, check that, for these variables, no storage locations which are modified by one task are modified or used in a potentially overlapping section of another task. In a few hours, we were able to locate a data sharing error in a large code. To do so requires understanding the synchronization structure, but the work being performed only needs to be inspected in terms of modifying and using storage locations.

## 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 èqually true in other environments.

To multitask or not? Multipocessor 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 propect 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 mutiuser 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 templete (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.

## 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 benefitted 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.

## References

1. G. Andrews and F. Schneider, *ACM Comput. Surv.* **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, *ACM Comput. Surv.* **5** (1973), 31.
4. G. J. Blair, "Reentrancy and Multitasking on Cray Computers," LLNL Report UCID-30199, May 15, 1984.
5. P. Brinch Hansen, *ACM Comput. Surv.* **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, *NMFECC Buffer* **8** (Sept. 1984).
8. K. Fong, *NMFECC Buffer* **8** (Oct. 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, Englewood Cliffs, N.J., 1982.
14. P. Brinch Hansen, "The Architecture of Concurrent Programs," Prentice–Hall, Englewood Cliffs, N.J.,1977.
15. R. C. Holt *et al.*, "Structured Concurrent Programming," Addison–Wesley, Reading, Mass., 1978.