CRAY RESEARCH, INC.

CRAY® COMPUTER SYSTEMS

CRAY X-MP MULTITASKING PROGRAMMER'S REFERENCE MANUAL

SR-0222

RECORD OF REVISION

PUBLICATION NUMBER SR-0222

Each time this manual is revised and reprinted, all changes issued against the previous version are incorporated into the new version and the new version is assigned an alphabetic level.

Every page changed by a reprint with revision has the revision level in the lower righthand corner. Changes to part of a page are noted by a change bar in the margin directly opposite the change. A change bar in the margin opposite the page number indicates that the entire page is new. If the manual is rewritten, the revision level changes but the manual does not contain change bars.

Requests for copies of Cray Research, Inc. publications should be directed to the Distribution Center and comments about these publications should be directed to:

CRAY RESEARCH, INC.
1345 Northland Drive
Mendota Heights, Minnesota 55120

Revision	Description
	February 1984 - Original printing.
A	January 1985 - This rewrite brings the publication into agreement with the Cray operating system COS version 1.14. All previous versions are obsolete.
B	March 1986 - This rewrite brings the publication into agreement with the Cray operating system COS version 1.15. It incorporates new material describing operation under the Cray operating system UNICOS, release 1.0, and includes microtasking, a new multitasking feature. All previous versions are obsolete.
C	October 1986 - This rewrite brings the manual into agreement with COS version 1.16 and UNICOS version 2.0. All trademarks are now documented in the record of revision. This printing obsoletes all previous versions.
D	July 1987 - This rewrite supports UNICOS version 3.0 and COS version 1.16. The manual has been reorganized and substantially rewritten for this release.

PREFACE

This manual is a guide for programmers and analysts who have an interest in producing software that can be multitasked during execution on Cray computer systems. It describes the multitasking features and associated concepts provided with the Cray operating systems COS and UNICOS on CRAY X-MP computer systems. The manual tells you how to use the features and how to produce executable programs that generate correct results.

It is assumed that you are familiar with the contents of the COS Version 1 Reference Manual, publication SR-0011, or the UNICOS User Commands Reference Manual, publication SR-2011. You are also expected to be experienced in coding Cray Fortran, using either CFT or CFT77. The Fortran (CFT) Reference Manual, publication SR-0009, and the CFT77 Reference Manual, publication SR-0018, describe these two versions of Fortran.

For multitasking on a CRAY-2 computer system, see the CRAY-2 Multitasking Programmer's Manual, publication SN-2026.

The following Cray Research, Inc. (CRI) publications also contain information useful to programmers developing multitasking software on Cray computer systems:

SR-0000	CAL Assembler Version 1 Reference Manual
SR-0012	Macros and Opdefs Reference Manual
SG-0056	Symbolic Interactive Debugger (SID) User's Guide
SR-0060	Pascal Reference Manual
SR-0066	Segment Loader (SEGLDR) Reference Manual
SR-0113	Programmer's Library Reference Manual
SR-0146	COS Performance Utilities Reference Manual
SR-2003	CAL Assembler Version 2 Reference Manual
SR-2014	UNICOS File Formats and Special Files Reference Manual
SG-2016	UNICOS Support Tools Guide
SR-2040	UNICOS Performance Utilities Reference Manual

CONTENTS

5. MACROTASKING (continued)

FIGURES

FIGURES (continued)

TABLES

NEW FEATURES

This sheet describes the new features in CRAY X-MP multitasking for UNICOS 3.0.

The **mtdump** program, which lets you display a log of events that occur during the execution of a multitasked program and print them in any of several formats, is now available under UNICOS as well as COS.

The barrier routines offer you another synchronization method for macrotasking. For instance, a barrier can be set up in a subroutine that is called by multiple tasks. Each task will stop when it reaches the barrier until all tasks have arrived at the same point. This feature is not available under COS 1.16.

In microtasking, new DO GLOBAL directives let you both vectorize and microtask an innermost DO loop. The facility automatically divides groups of 64 iterations each among the available processors. This feature is not available under COS 1.16.

The microtasking CMIC$ MICRO directive is no longer required in subroutines that have either a PROCESS directive or a DO GLOBAL directive. This is not the case under COS 1.16; the CMIC$ MICRO directive is still required.

The estimated timings for multitasking routines have changed somewhat. Appendix C details the new timings.

1. INTRODUCTION

This manual describes multitasking on CRAY X-MP computer systems under COS and UNICOS. Multitasking is a mode of operation, in a multiprocessor computer, that provides for execution of two or more parts of a single program in parallel. An efficiently multitasked program executes in less wall-clock time, when multiple processors are available, than a program that is not multitasked.

As you will learn, Cray Research offers several techniques for making use of multiple processors. When choosing among these techniques, you must make trade-offs between the overall performance improvement that can be achieved and the level of effort that you are willing to put into modifying the application.

In general, the easiest way to multitask a program is at the level of a Fortran DO loop, but the performance gains are limited to the scope of that loop. The most efficient use of multitasking is at a very high level within the program, but this requires a detailed understanding of both the structure and algorithm of the candidate program. Fortunately, you can combine techniques, which allows you to experiment or choose the combination that best suits your application and resources.

This manual includes the following:

- Concepts related to multitasking
- Descriptions of features
- Procedures and advice for programmers producing multitasked code from existing code

This document assumes that the code to be multitasked is running on a CRAY X-MP computer system with multiple processors, although multitasked code can be run on single-CPU systems for purposes of program development and debugging. Appendix A provides information for running multitasked code on single-processor CRAY X-MP and CRAY-1 computer systems. For multitasking on a CRAY-2 computer system, see the CRAY-2 Multitasking Programmer's Manual.

1.1 MULTITASKING TRADE-OFFS

When multitasking a program, you surrender the overhead time incurred by calls to the multitasking routines while gaining performance by applying more than one processor to the program.

The theoretical gain that can be achieved from multitasking on a dedicated system is the wall-clock time the program requires without multitasking, divided by the number of processors. On a CRAY X-MP/4 computer system, with four processors, the greatest wall-clock speedup due to multitasking is a factor of 4.

In practice, however, a speedup factor equal to the number of processors is not quite attainable. In extreme cases, multitasking can actually increase a program's execution time if the multitasking overhead decreases performance more than parallel execution improves it. This is a situation you will want to predict before investing too much time and effort. There are some factors that limit the maximum improvement for a program:

- Not all parts of a program can be divided into parallel tasks. Many algorithms do not have a parallel structure or have only a portion that is parallel.

- The parts that can be multitasked may have dependencies on one another that result, at run time, in one or more tasks having to wait until others complete some operation. During this wait time, the waiting tasks do not contribute to parallel execution, and the CPUs may not be readily available to other jobs in a multiprogramming environment.

- Use of the multitasking features incurs a certain amount of overhead that increases the execution time but does not directly increase the computation rate. The more these features are used, the greater the overhead.

The initial implementation of multitasking at CRI, called macrotasking, was directed toward long-running, large-memory programs running in a dedicated environment. Macrotasked programs can be run in a batch environment, but improvement in execution time can vary greatly from run to run, depending on other activity in the system. Total system throughput may decrease if the increased CPU time used by macrotasked programs reduces the time available to other programs. (A batch job that requires all of the memory available to a single user effectively executes in a dedicated environment. In such a case, you should consider multitasking to make use of all processors.)

Macrotasking can make programs difficult to test and debug. When two or more parts of a program are executed simultaneously, timing errors can arise. These errors may not be reproducible, and currently available facilities to help analyze or prevent such timing errors are limited.

Converting a program for macrotasking requires more analysis than does converting for vectorization. The CFT compiler and the CFT77 compiler automatically perform vectorization, which can give performance improvements over scalar code as good as, or better than, multitasking. Modifications can increase the amount of code that can be vectorized, but these tend to be small changes and, in the case of CFT, localized to inner DO loops. The majority of the modifications to vectorized code are safe; CPU time rarely increases, and answers remain correct.

Because macrotasking is a more recent and complex enhancement than vectorization, there are fewer analysis aids to assist you in producing macrotasked code. Further, modifications for macrotasking may involve larger segments of code than do vectorization modifications, because macrotasking is often applied on a subroutine basis rather than a DO-loop basis. You must appreciate the overhead costs of macrotasking and be willing to enforce the rules necessary for producing correct results to benefit from the significant performance improvements of macrotasking.

Following the implementation of macrotasking, other approaches to multitasking were investigated. One of the most successful of these is microtasking, described in section 4. Microtasking allows some of the following improvements over macrotasking:

- Tasks can be much smaller (generally a set of nested loops), simplifying the programmer's job of conversion.

- Processors are used for periods when other jobs are not using them. This has the potential to increase total system throughput in a batch production environment by using otherwise idle CPU cycles.

- Synchronization overhead is very low. When run on a single processor, code modified for microtasking runs nearly as fast as the original code.

Microtasking is invoked with processor directives in Fortran source code. The preprocessor, PREMULT, outputs the necessary library calls. Microtasking and macrotasking can be used in the same application, if appropriate. Many of the concepts and approaches described in this manual apply to both.

The third multitasking technique is that of automatic partitioning by the Fortran compiler, CFT77. With this feature, the compiler automatically recognizes language constructs that lend themselves to be multitasked. In addition, the compiler accepts directives, similar to those processed by PREMULT, to provide user-controlled multitasking. The first implementation of this feature will be in CFT77 version 2.0.

Multitasking is valuable in certain applications, and you should consider it as a possible performance enhancement, evaluating the ratio of costs to benefits for each application.

1.2 MULTITASKING OVERVIEW

Multitasking, with each of the three techniques, occurs completely within a user job. The operating system, COS or UNICOS, allows a program to create separate tasks that are then scheduled onto separate processors. Code in library subroutines or generated by PREMULT or CFT77 manage the tasks created by the program and make the necessary operating system calls.

The multitasking techniques described here are supported on both COS and UNICOS, and the user interface is identical on the two systems. The changes made on one system for multitasking can be carried over to another system.

Both operating systems support the same library routines and preprocesser directives for multitasking. However, because of timing differences within the system, the efficiency of macrotasking may vary between the two systems.

1.2.1 COS

COS provides for multitasking within *job steps*. Each control statement in a job control language (JCL) file is a job step, although only a job step that executes code compiled from a user program normally makes use of multitasking. The job steps themselves are executed sequentially. A program executing in a job step can create additional tasks, bringing about multitasking. A multitasked job step is not complete until all tasks within the job step are complete.

The following example shows the lifetimes of different tasks for a job that builds and runs a program partitioned into three tasks. All but the MTPROG job step use only one task; of course MTPROG probably requires the most execution time. The MULTI control statement is required to access the libraries necessary for multitasking.

Example:

	Task 1	Task 2	Task 3
JOB,JN=TMULT...	X		
ACCOUNT,AC=...	X		
MULTI.	X		
CFT,ALLOC=STACK...	X		
SEGLDR,CMD='ABS=MTPROG'.	X		
ACCESS,DN=DATA,PDN=DATA1,...	X		
MTPROG.	X	X	X
SAVE,DN=OUT,PDN=OUT1,...	X		

No CRI software products or utilities have been internally multitasked. Successive compilation steps, for example, do not execute in parallel.

A COS job that is multitasked can run on the same system with jobs that are not multitasked. Although the wall-clock time and the order of execution of tasks within the job may change, a properly multitasked job should see no change in results.

1.2.2 UNICOS

UNICOS provides for multitasking within user programs. A user program can create additional tasks within its own memory image, thus bringing about multitasking. A multitasked user program is not complete until all tasks within the program are complete.

1.3 CONVENTIONS

Throughout this manual, a variety of typefaces, special characters, and formats are used to indicate special terms and their use. They are as follows:

Convention	Description
Italic	Italic indicates the following: • Within a syntax representation, italics represent variable information to be supplied by you. • In text, italics indicate either the first use of a term being defined or that a word is being used to represent itself rather than its meaning.
Boxes	Boxes enclose syntax representations, usually indicating a call to a library routine and often including italicized variable names.
UPPERCASE	In text and in syntax representations, uppercase words indicate Fortran keywords, such as *COMMON* and *TASK COMMON*, or other words that appear in code, such as *MTPROG*.
boldface	In text, boldface identifies UNICOS commands, parameters, and files.

An additional convention regards the synonymous use of the terms *dataset* and *file* in this publication. The COS term *dataset* usually means the same thing as the UNICOS term *file*, except a dataset can contain more than one file.

1.5 READER COMMENTS

If you have any comments about the technical accuracy, content, or organization of this manual, please tell us. You have several options that you can use to notify us:

- Call our Technical Publications department directly at (612) 681-5729 during normal business hours
- Send us UNICOS or UNIX electronic mail at this address:

  ihnp4!cray!publications or **sun!tundra!hall!publications**

- Use the Reader Comment form at the back of this manual
- Write to us at the following address:

  Cray Research, Inc.
  Technical Publications Department
  1345 Northland Drive
  Mendota Heights, Minnesota 55120

We value your comments and assure a prompt response.

2. CONCEPTS

This section defines the concepts and terminology of multitasking as they are applied by CRI.

These terms are far from standard within the industry. In fact, the terminology differs somewhat between macrotasking and microtasking. The concepts described here apply to both macrotasking and microtasking in some degree, although macrotasking examples are used most often to illustrate them. Terms specific to either microtasking or macrotasking are treated in sections 4 and 5, respectively.

2.1 PARALLELISM

As used in this manual, *parallel* refers to the manner in which software processes are executed on a computer. Jobs, job steps, programs, and parts of programs are parallel if they are processed simultaneously (or nearly so) rather than sequentially. *Parallel processing* is therefore the simultaneous processing of two or more segments of code.

The types of software processes that are executed in parallel define the levels of parallelism:

Level	Software Process
1	Independent jobs, each job having a CPU
2	Job steps: related parts of the same job
3	Routines and subroutines
4	Loops
5	Statements

The higher the level number, the smaller the size or *granularity* of tasks.

Vector processing is parallel processing of loop iterations (level 4). CFT77 and CFT schedule generated instructions in a manner that exploits the independence and different speeds of the hardware functional units; this leads to parallel execution of different statements (level 5).

2.2 MULTIPROGRAMMING

Multiprogramming is a property of the operating system that permits overlapping and interleaving the execution of more than one program. Many computer systems use multiprogramming to make the most efficient use of a single CPU. In this mode, several processes are ready to run, and if I/O delays one process, the system immediately schedules another process to run on the CPU. In contrast, a system in dedicated mode has only one process ready to run, and any delays leave the CPU idle. The processor resource can consist of more than one CPU; each CPU could be shared by several software processes.

Example:

COS and UNICOS are both multiprogramming operating systems. The processor resource is one CPU, and the software processes are jobs. The Job Scheduler manages sharing within the operating system by assigning priorities to jobs and allocating CPU time, a slice at a time, to different jobs. Figure 2-1 shows this type of multiprogramming.

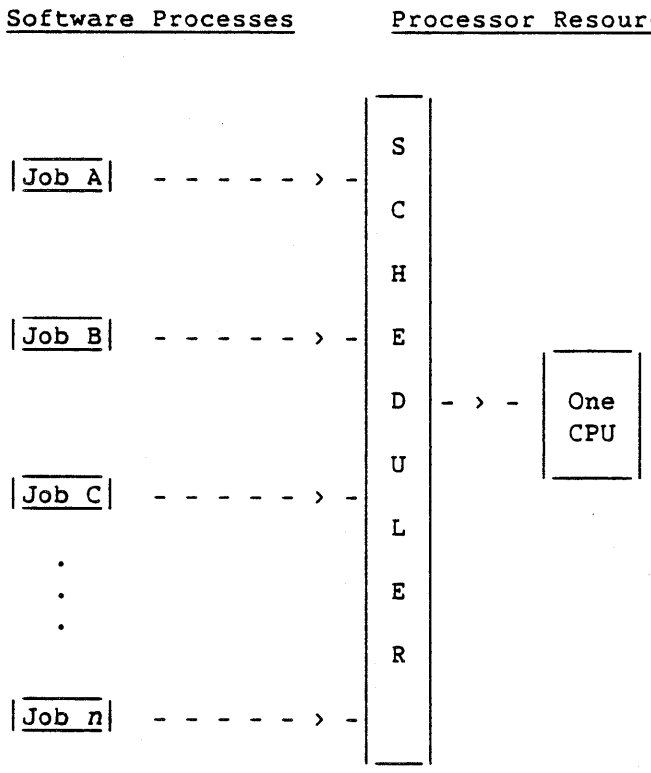


Figure 2-1. Multiprogramming

2.3 MULTIPROCESSING

Multiprocessing is a property of the hardware in which two or more CPUs are available. The processors can all work simultaneously without adversely affecting each other.

For example, four independent jobs can be run in parallel on a CRAY X-MP/4 computer system under COS or UNICOS. A series of jobs running in this mode is called a *job stream*, one stream on each processor. The job is the scheduling unit of the system, and four processors are scheduled in a multiprogramming mode. Truly independent jobs do not affect each other, but two (or more) jobs using the same dataset can interfere with each other and thus are not independent.

This example of independent uniprocessing exploits parallelism at level 1. Independent uniprocessing enhances system throughput over single-processor configurations, though individual jobs receive smaller turnaround time for the same system workload.

Applying more than one processor to a single job implies that the job has software processes (parts) that can be executed in parallel. Such a job can be logically or functionally divided to allow two or more parts of the work to execute simultaneously (that is, in parallel). An example of this is a weather modeling job in which the northern hemisphere calculation is one part and the southern hemisphere is another part. Another example of a job that can be functionally divided is a program having a sort operation on a database that can be run independently of a formatting operation on previously processed data.

Distinct code segments need not be involved. The same code could run on multiple processors simultaneously, with each processor acting on different data.

2.4 TASK

A *task* is a unit of computation that can be scheduled, with instructions that are processed in sequential order. It is a software process that can consist of one or more subroutines. Under COS and UNICOS, a job is a task that can *spin off* other tasks to run in parallel with it. That is, it is a software process that can call other software processes (such as subroutine tasks) that will be executed simultaneously with it.

To take advantage of a multiprocessing operating system, you must be able to divide a job into two or more tasks, providing parts of the job that can be separately scheduled and run in parallel on more than one processor.

A task is a uniquely named process that can have code and data areas in common with (or even identical to) other tasks from the same job. The code executed by a task is a subroutine. The same work can be performed by calling the subroutine or by starting a task to execute the subroutine. The difference is that a subroutine call causes the work to be performed immediately; while in the task, the system schedules and performs the work independently and in parallel with other program tasks.

NOTE

The term *task* in CRI publications refers to several types of software entities. Except as otherwise indicated, any reference to task in this manual uses the preceding definition, which corresponds to the concept of *library task* in other CRI publications.

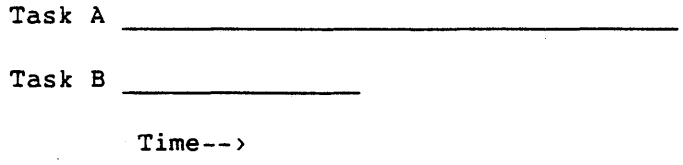
2.5 MULTITASKING

Multitasking is the structuring of a program into two or more tasks that can execute concurrently. With macrotasking (see section 5, Macrotasking), multitasking is supported only for subprograms (level 3 parallelism). With microtasking (see section 4, Microtasking), multitasking is supported for subroutines, functions, loops, or possibly statements or sets of statements.

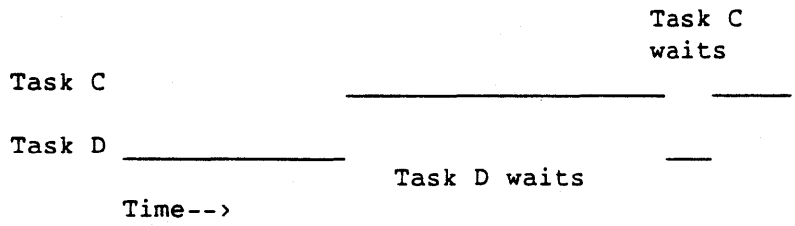
In a multitasking environment, the tasks and data structure of a job must be such that the tasks can run independently of each other, either synchronously or asynchronously. There is no certainty that more than one processor will be able to work on the tasks of a given job, that the tasks will execute in any particular order, or that a particular task will finish first. The availability of processors and the order of execution and completion of tasks are functions of the scheduling policies of the library and operating system; multitasking is *nondeterministic* with respect to time.

Tasks must be made deterministic, however, with respect to results. The key to a successful multitasked program is to precisely define and add the necessary communication and synchronization mechanisms between parallel tasks and to provide for the protection of shared data.

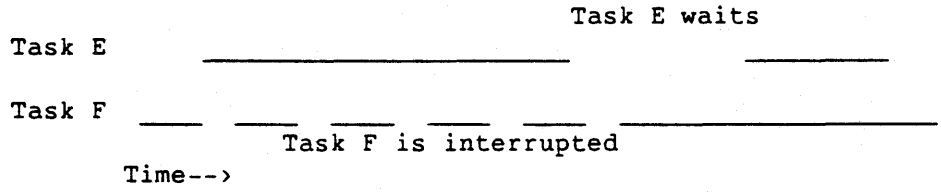
The following example is a simple case in which two tasks execute without interruption on two processors (the solid lines indicate a CPU engaged and executing code):



In the next example, only one processor is available, and tasks C and D must share it; this shows multitasking on a machine with one processor.



In the third example, two tasks share two processors. At different points throughout execution two, one, or zero processors are assigned to the job. Further, there is no indication of which physical processor is assigned to which task; this assignment is transparent to the program.



2.6 SCOPE

The *scope* of a variable is the region of a program in which the variable is defined and can be referenced. Outside of a variable's scope, the variable is not defined, and references to the variable's name either refer to another variable of the same name (as in Fortran) or are treated as an error condition (as in Pascal or CAL) if not otherwise declared.

Each task consists of executable instructions and a well-defined set of data upon which the instructions act. The set of all variables that a task can reference can be divided into two subsets: one being the variables *private* to the task, and the second consisting of the variables *shared* between the task and at least one other task. Private variables are defined for one task and are accessible only by that task, while shared variables are defined for and accessible by several tasks.

While the terms *private* and *shared* refer to the scope of data in relation to tasks, the terms *global* and *local* refer to the scope of data in terms of program units, such as subroutines. Global variables can be accessed by multiple program units. Local variables can be accessed only within a single program unit.

A variable used in multitasking has a scope in relation to both its task and its program unit. Such a variable may be, for example, global to a set of subroutines but private to a single task.

CFT77 and CFT, like many other Fortran compilers, guarantee the value of local variables only for the lifetime of the subroutine containing them. Global variables, which are named in COMMON blocks, are guaranteed for the lifetime of the entire program. In a Fortran program that is not multitasked, this distinction can often be ignored, because the local variables are usually assigned to a fixed location in Central Memory to improve performance. In a multitasked program, the location of variables local to subroutines and private to tasks can change and the memory space that they occupy can be reused. This makes the distinction between variables private to tasks and variables shared between tasks important to understand and respect.

In the subset of shared variables, you must include all variables used for communication between tasks and variables on which more than one task works (for example, a large array in COMMON for multiple tasks).

Variables used in the internal functioning of a task (for example, loop indices and variables controlling the flow of execution) must be included in the subset of privately defined variables.

You should take special care when multitasking tasks that have identical code (that is, the same subroutines are associated with different tasks). Certain variables may need to be accessed by all the subroutines within a task but be private to that task, even though another task has identical code. The CRI Fortran extension, TASK COMMON, satisfies this intermediate scope requirement (see subsection 9.6, Task Common). This intermediate scope can create confusion, because the scope of a variable is normally determined by the division of code into program units, but when you divide code into tasks, data that is global among subroutines should not necessarily be shared among tasks. A new level of scope has been defined and along with it comes the necessity of making new distinctions between variables.

2.7 CRITICAL REGION

A *critical region* is a segment of code that accesses a shared resource. This resource can be Central Memory, I/O files, subroutines, or anything else that is shared by the tasks in a job. (Most examples of critical regions in this manual relate usually to shared memory, though the concepts and techniques apply equally well to any shared resource.)

For example, indeterminate results can arise when more than one task simultaneously reads from and stores into shared memory locations. Neither task can be sure that the data it is reading is as expected, nor that the area of memory to which data is stored is ready to be overwritten.

As an example, consider the following subroutines (each of which is a task) running in parallel:

```
      SUBROUTINE MTASK1
      COMMON /COMA/AAA
      AAA = 0.
      (start task MTASK2)
      ...
      AAA = AAA + 1.
      ...
      (wait for completion of MTASK2)
      ...
      END

      SUBROUTINE MTASK2
      COMMON /COMA/AAA
      ...
      AAA = AAA + 1.
      ...
      END
```

Variable AAA is shared, because both task MTASK1 and task MTASK2 could change it simultaneously.

You must ensure that critical regions of code are monitored if the program modules containing them are to run in parallel. A way that you can accomplish this monitoring is to have each code segment set a lock when it enters a critical region. In effect, the task sets up a flag to indicate that the shared variables are being used. This system works only if all other tasks that can run in parallel check the lock before they enter a corresponding critical region. The monitoring operation consists of the following steps:

1. Test to see whether the lock is set.

2. If the lock is set, wait until it is cleared and then go to step 3. If the lock is clear, go immediately to step 3.

3. Set the lock and enter the critical region.

4. Clear the lock when exiting the critical region.

In most implementations of this feature, including the CRI implementation, a task executing this operation waits in step 2 if the lock is set, until another task leaves the critical region and clears the lock.

A program in which all instances of a critical region are successfully monitored is said to be implementing mutual exclusion within the critical region. That is, if one task is in the region, all others are excluded. This is called *singlethreading*.

Because a task unable to enter a critical region is forced to wait, it is important to keep the length of critical regions (in execution time) to a minimum. This goal must be balanced against the cost of the locking operation. A job that has overly large critical regions can have numerous tasks waiting for entry; but a job with too many, overly small critical regions can incur a high overhead penalty. The following examples demonstrate critical regions.

Example 1:

```
      SUBROUTINE MTASK1
      COMMON /COMA/AAA
      AAA = 0.
      (start task MTASK2)
      ...
C        BEGIN CRITICAL REGION
      AAA = AAA + 1.
C        END CRITICAL REGION
      ...
      (wait for completion of MTASK2)
      ...
      END

      SUBROUTINE MTASK2
      COMMON /COMA/AAA
      ...
C        BEGIN CRITICAL REGION
      AAA = AAA + 1.
C        END CRITICAL REGION
      ...
      RETURN
      END
```

Example 2:

```
      SUBROUTINE MTASK1
      DIMENSION A(1000),B(1000)
      COMMON/BLOCK/J,A,B,N
      INTEGER JLOCAL
      ...
C        BEGIN CRITICAL REGION
      JLOCAL = J + N
      J = JLOCAL
C        END CRITICAL REGION
      ...
      DO 10 I = 0,N-1
        A(I+JLOCAL) = B(I+JLOCAL)
10    CONTINUE
      ...
      END

      SUBROUTINE MTASK2
      DIMENSION A(1000),B(1000)
      COMMON/BLOCK/J,A,B,N
      INTEGER JLOCAL
      ...
C        BEGIN CRITICAL REGION
      JLOCAL = J + N
      J = JLOCAL
C        END CRITICAL REGION
      ...
      DO 10 I = 0,N-1
        A(I+JLOCAL) = B(I+JLOCAL)
10    CONTINUE
      ...
      END
```

In the previous example, references to J in MTASK1 and MTASK2 are critical regions and must be monitored to ensure that each loads a different value of J.

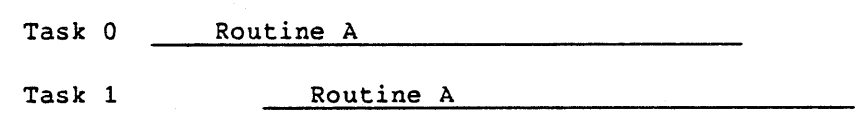
2.8 REENTRANCY

Reentrancy, also called *multithreading,* is a property of a program module that allows one copy of the module to be used by more than one task in parallel. A mechanism re-creates the routine's local environment each time the routine executes. That is, local variables and control indicators are assigned independent storage locations each time the routine is invoked.

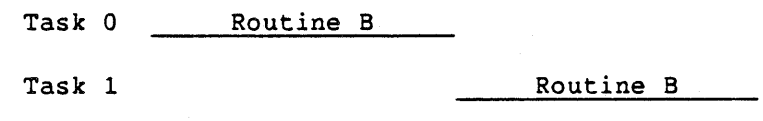
Not all program modules in a multitasked program need be used in a reentrant sequence. For example, a module that is executed only once during the lifetime of the program is a *nonreentrant* code segment. A module that is in a critical region, so that no more than one task at a time can execute it, is a *serially reusable* code segment.

Example:

Routine A is reentrant.



Routine B is serially reusable.



Serially reusable code is necessary if a task requires access to a resource (typically memory) that is in COMMON with other tasks. Serial execution is necessary to prevent two or more tasks from simultaneously altering the same data item.

In some situations, simultaneous processing of one data item by two (or more) tasks may not cause problems. For example, if both tasks are merely referring to the item and not altering it, no inconsistencies arise. Many times, multiple tasks redefine as well as reference common data. In such a situation, you must ensure that only one task is using and altering the data at any one time. You must design each task to ensure serial reusability of the code segment dealing with the COMMON data.

To guarantee serial reusability of the code segment, use the locking operation described in subsection 2.7, Critical Region. You can do this in the following three ways:

- If the program is designed so that no attempt is ever made to reenter the code segment, it needs no special treatment.
- If the entire subroutine is nonreentrant (as with Fortran code compiled with ALLOC=STATIC; see section 5, Macrotasking), you must treat all calls to the subroutine as critical regions and lock them.

- If the entry sequence is reentrant (as with CFT77 or CFT code compiled with ALLOC=STACK), any nonreentrant parts of the subroutine can be locked within the subroutine.

Examples:

1. Subroutine SERIAL is totally nonreentrant (compiled with the STATIC compilation option).

```
SUBROUTINE MTASK
...   (declarations, followed by code)
CALL LOCKON (LSERIAL)
CALL SERIAL
CALL LOCKOFF (LSERIAL)
...   (code)
RETURN
END
```

2. Subroutine SERIAL has a reentrant entry sequence (compiled with the STACK compilation option).

```
SUBROUTINE SERIAL
...   (declarations, no code)
CALL LOCKON (LSERIAL)
...   (code)
CALL LOCKOFF (LSERIAL)
RETURN
END
```

Regardless of the reentrancy of a program module, any critical regions within it must still be monitored and locked. For example, consider the following two modules (both compiled with the STACK compilation option):

```
SUBROUTINE SERIAL
...   (declarations, no code)
CALL LOCKON (LSERIAL)
...   (code)
CALL LOCKON (LCRIT1)
...   (critical region)
CALL LOCKOFF (LCRIT1)
...   (code)
CALL LOCKOFF (LSERIAL)
RETURN
END
```

```
SUBROUTINE PARALLEL
...   (declarations)
CALL LOCKON (LCRIT1)
...   (critical region)
CALL LOCKOFF (LCRIT1)
...
RETURN
END
```

Even though SERIAL in the previous example is serially reusable, it must separately protect the critical region with LCRIT1, because PARALLEL might be executing simultaneously. LSERIAL can be used to protect both the critical region and the subroutine SERIAL, but this may have the disadvantage of increasing the critical region's size. (PARALLEL would be locked out for the entire time SERIAL is executing, not just the time SERIAL is inside the critical region.)

Appendix F, Design Description, describes the CRI implementation of multitasking, which uses a stack mechanism.

2.9 LOAD BALANCING

Load balancing is a technique ensuring that each of the processors involved in a job does approximately the same amount of work. All work that can be done in parallel is divided evenly among processors. There are two types of load balancing: static and dynamic. Because microtasking performs dynamic load balancing automatically when processes have small granularity, load balancing is generally a concern in macrotasking applications.

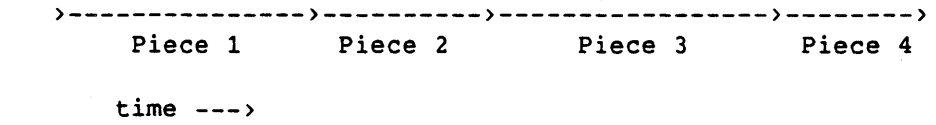
Static load balancing is possible when you can determine ahead of time the amount of work involved in each piece of a job. You then define parallel tasks, each of which runs in a similar amount of time (see subsection 3.5.2, Load Balancing).

A program whose pieces have unknown workloads needs dynamic load balancing. Because it is impossible to predict the amount of time that a given piece requires, you should construct tasks that continually look for and execute the next piece of work (see subsection 3.5.2, Load Balancing).

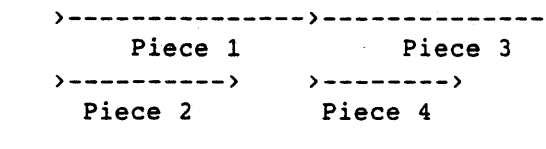
If all the work involved in a job can be done in parallel on n processors and the load is balanced among them, the wall-clock time for the multitasked job can approach $1/n$ of the wall-clock time for the job run on one processor.

Example:

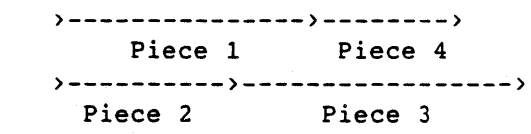
One task (serial code):



Two tasks (partially balanced code):



Two tasks (better balanced code):



2.10 SYNCHRONIZATION

Synchronization, as used in multitasking, is the method of coordinating the steps within tasks that can be run in parallel. Coordination ensures that initial conditions for a task are met or that output from a task is ready to be used.

A synchronization point is a point in time when a task receives the go-ahead to proceed with its processing. That is, whatever the task is awaiting has happened, and a signal has been sent to and received by the waiting task.

The CRI macrotasking implementation provides four synchronization mechanisms:

- Events, which provide a general way of signaling the occurrence of some programmer-defined event. Tasks can wait for events, post events (that others may be waiting for), or clear events (reset them).

- Locks, if initialized to the locked state, can be used by a signaling task by calling LOCKOFF and by a waiting task by calling LOCKON.

- A task can wait for another task to complete execution. This could be viewed as a higher-level function based on the event mechanism (where the event is a task completion and is posted by the system).

- Barriers, which keep tasks waiting at a specified point until all tasks reach that point.

(Microtasking has a synchronization feature similar to locks. This feature uses the GUARD and ENDGUARD directives.)

Synchronizing tasks works only if all tasks perform their respective parts of the required communication. One task must signal the important occurrence; another task or tasks must wait for the signal, receive it, and clear the signaling device. The following example shows two tasks using events and critical regions:

Example:

Two tasks:

```
Task 0   !-----W          --(  ---)-----P----!
Task 1   !---------------P-(--)----W   ---------!
```

Symbol	Meaning
W	Wait for event occurrence
P	Post event occurrence
(	Request to enter critical region
)	Leave critical region

The periods each task loses to the synchronization and locking mismatches increase the total run time of each task. The load-balancing technique discussed previously should take such possible synchronization delays into account.

2.11 DEADLOCK

Deadlock is a condition in which locks and synchronization mechanisms have been misused to the extent that a task is waiting for an event that can never happen.

As a simple case, consider the following incorrect code segment:

```
      DO 10 I=1,N
        CALL LOCKON(LOCK1)
        ...
  10  CONTINUE
      CALL LOCKOFF(LOCK1)
```

A task executing this code successfully locks LOCK1 in the first iteration but waits forever in the second iteration. The call to LOCKOFF was intended to be within the loop.

A more frequently encountered form of deadlock is when two tasks wait for each other to complete some action. For example, consider two tasks each using two locks that are set in a different order. In such a case, each task might set one lock and wait for the other lock to be cleared. Such a situation would not necessarily occur in every run, because it is tied to the timing of the two tasks.

A deadlock need not initially involve all tasks in the job. Even if only a subset of tasks deadlocks initially, the other tasks will either complete or will wait themselves. Eventually, all active tasks in the job are deadlocked.

Deadlock detection is recognizing a deadlock situation after the deadlock occurs. *Deadlock prevention* requires conventions or rules to ensure that a deadlock does not occur. For example, you can define a rule specifying that any task needing more than one lock must set the locks in alphabetical order. This prevents deadlock, although at the possible cost of enlarging a critical region. Deadlock detection is a function of the system software; you are generally responsible for deadlock prevention.

2.12 AMDAHL'S LAW AND THEORETICAL SPEEDUP

Multitasking is an optimization that changes the apparent execution time of code segments. The overall execution time of a multitasked program behaves according to *Amdahl's Law* based on the percentage of time multitasked. The following paragraphs define Amdahl's Law.

Let T1 be the execution time of a nonmultitasked program. If multitasking is applied to a fraction (f) of the original execution time, the theoretical execution time (assuming no overhead or delays) is the time to do the sequential portion (Ts) plus the time required to perform the multitasked part (Tm). These quantities are a function of the original execution time, the number of processors (P) and f.

T1 = Original execution time
f = Fraction of T1 multitasked
Ts = (1-f)*T1 = Time of sequential part
Tm = (f/P)*T1 = Time of multitasked part
P = Number of processors

The theoretical speedup attainable with P processors, S(P,f), is a ratio of the original execution time to the total execution time of the multitasked program, or Amdahl's Law. Figure 2-2 shows this theoretical speedup as a function of the fraction of the program getting multitasked.

$$S(P,f) = \frac{T1}{Ts + Tm} = \frac{T1}{T1 * ((1-f) + (f/P))}$$

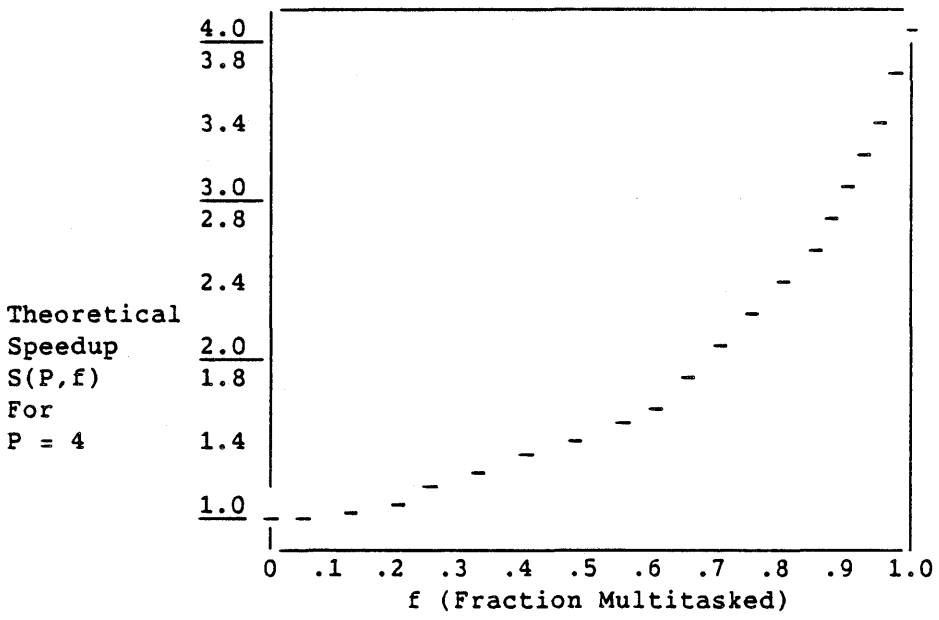


Figure 2-2. Amdahl's Law Curve

Figure 2-2 shows the Amdahl's Law Curve, a plot of speedup versus *f*, for a four-processor system.

Table 2-1 shows the theoretical speedup for different values of P and f.

Table 2-1. Theoretical Speedup

f	P=1	P=2	P=3	P=4	P=8	P=16	P=32	P=64	P=infinity
1.00	1.00	2.00	3.00	4.00	8.00	16.00	32.00	64.00	infinity
0.99	1.00	1.98	2.94	3.88	7.48	13.91	24.43	39.26	100.00
0.98	1.00	1.96	2.88	3.77	7.02	12.31	19.75	28.32	50.00
0.97	1.00	1.94	2.83	3.67	6.61	11.03	16.58	22.14	33.33
0.96	1.00	1.92	2.78	3.57	6.25	10.00	14.29	18.18	25.00
0.95	1.00	1.90	2.73	3.48	5.93	9.14	12.55	15.42	20.00
0.94	1.00	1.89	2.68	3.39	5.63	8.42	11.19	13.39	16.67
0.93	1.00	1.87	2.63	3.31	5.37	7.80	10.09	11.83	14.28
0.92	1.00	1.85	2.59	3.23	5.13	7.27	9.19	10.60	12.50
0.91	1.00	1.83	2.54	3.15	4.91	6.81	8.44	9.59	11.11
0.90	1.00	1.82	2.50	3.08	4.71	6.40	7.80	8.77	10.00
0.75	1.00	1.60	2.00	2.28	2.91	3.37	3.66	3.82	4.00
0.50	1.00	1.33	1.50	1.60	1.78	1.88	1.94	1.97	2.00
0.25	1.00	1.14	1.20	1.23	1.28	1.31	1.32	1.33	1.33
0.10	1.00	1.0	1.07	1.08	1.09	1.10	1.11	1.11	1.11
0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 2-1 suggests the following points:

- The small entries in the lower part of table 2-1 show that significant speedups are not possible unless significant portions of a program are multitasked.† For example, if 50% of the time in a code is multitasked on four processors, the best possible speedup is 1.6. This demonstrates that multitasking over P processors does not result in a speedup factor of P.

- The upper rows show that, for a fixed percentage of multitasked execution time, the speedup does not increase as fast as the number of processors. Indeed, the speedup converges to a finite limit for P=infinity because of the fixed percentage of time not multitasked. For a large number of processors, execution time is dominated by nonmultitasked code. Speedup factors approaching the number of processors are possible, but for a larger number of processors, more of the code must be multitasked.

† With respect to Amdahl's Law, *portion* refers to execution time and not to physical size. If, for example, a 10-line DO loop in a 100-line program is responsible for 90% of the CPU time accumulated by that program, the portion that loop represents is 90%, not 10%.

Because multitasking itself adds serial overhead to a job, the actual speedup is less than the theoretical speedup. Multitasking 50% of a job on a four-processor machine probably produces an actual speedup of about 1.5 rather than the theoretical 1.6.

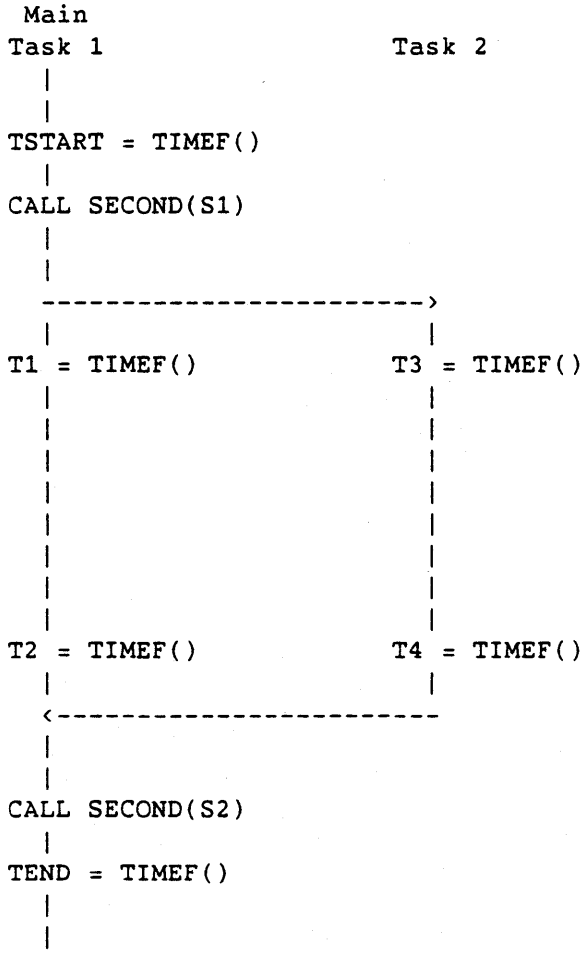
2.13 MEASURING TIME AND WORK

For a single CPU system used in a dedicated mode, work is often measured with CPU charges in time units. For computational jobs, the CPU time and the wall-clock time are frequently quite close, in which case no distinction between time and work is necessary.

The ability of several CPUs to run simultaneously creates a distinction between work and the time to complete the work. In a multitasked job, several CPUs may perform more work in less real time than would an equivalent program executed on a single processor. Routines are provided to measure time and work. The following routines measure time:

- TIMEF measures wall-clock time in milliseconds.
- SECOND measures work, calculating accumulated CPU charges for a job in seconds.

The following example shows how you can use these routines:



The difference (TEND-TSTART) measures the wall-clock time of the program in milliseconds, while S2-S1 measures the work (in CPU seconds) done by both tasks during this time. The differences (T2-T1 and T4-T3) each measure the wall-clock time in their respective tasks. No routine is currently available that measures work on a task basis.

For compute-bound jobs run on a dedicated two-processor system, (S2-S1) should be slightly greater than (T2-T1) plus (T4-T3). (TEND-TSTART) should be slightly greater than MAX(T2-T1, T4-T3). If the tasks do large amounts of I/O or if the processors are not dedicated but are rather switched by the system between many jobs, there is no predictable relationship between chargeable CPU seconds and wall-clock time.

3. MULTITASKING BASICS

Multitasking is a tool for improving the performance of programs. An understanding of how and when the tool should be used and what can be gained by using it is important. This understanding is based on concepts that previously may not have concerned the programmer but are now of fundamental importance.

3.1 GAINS WITH MULTITASKING

Multitasking does not reduce the number of CPU cycles necessary to execute the program. In fact, multitasking introduces an overhead that increases CPU time; therefore, the number of calls to the multitasking library must be minimized. You should exploit parallelism at the highest level possible to reduce the overhead.

Multitasking reduces the elapsed (turnaround) time of a single program executed in a monoprogramming (dedicated) environment. This major benefit of multitasking results from all processors being available to simultaneously consume the CPU time needed for the program. Production work profits from multitasking in this mode.

In a multiprogramming (batch) environment, the tasks of a multitasked job compete with other jobs in the system for the processor resources. The speedup of the multitasked job depends on how successful the job is in this competition. Job priority can favor multitasked jobs over other jobs for improved performance, and experimental and developmental work can profit from multitasking in this mode. Microtasked jobs, with their low overhead and use of otherwise idle processors, can improve throughput in a batch environment and enhance overall throughput of the system (see section 4, Microtasking).

3.2 COMPUTATIONAL AND STORAGE DEPENDENCE

The ability to execute tasks in parallel requires that each task be independent of other tasks. This independence comes in several forms, and tasks must be independent in all forms before they can be multitasked.

Multitasking, as supported by CRI, operates on the subroutine level, but you can often find sections of unconverted code at the loop level that can be broken into separate tasks. Such code, if determined to be independent, can be broken apart into separate subroutines to take advantage of multitasking. This subsection describes multitasking independent iterations of loops.

3.2.1 COMPUTATIONAL DEPENDENCE

Computational dependence is composed of data dependence and control dependence. These forms of dependence can be analyzed by looking at dependence graphs, which depict relationships between statements in a program. They are useful in understanding the constraints a program must impose to preserve orderly (deterministic) and correct results. Analyzing dependence graphs shows when the opportunity for vectorization and multitasking exists.

Dependence graphs are composed of nodes representing the statements in a program and directed arcs connecting nodes representing the ordering constraints. These graphs are concerned with assignment statements, assignments within loops, and I/O statements. The various types of dependencies between statements are described in the following subsections.

3.2.1.1 Data dependence

Data dependence is an ordering relationship between statements that use or produce the same data. The ordering must be preserved to generate correct results. Analysis of data dependence determines whether the statements can be vectorized or executed in parallel without violating this ordering. The types of data dependence are as follows:

- Flow dependence
- Antidependence
- Output dependence
- Unknown dependence
- I/O dependence

3.2.1.1.1 Flow dependence - Statement *s2* is flow dependent on statement *s1* if an execution path exists from *s1* to *s2* and if the same variable (scalar or array element) is assigned in statement *s1* and used in statement *s2*. A directed arc connects node *s1* to a flow-dependent node *s2* in a dependence graph.

Example:

Code		Dependence Graph
	DO 10 I = 1,N	*s1*
1	A(I) = B(I) + 5.	↓
2	C(I) = A(I) * 3.	
10	CONTINUE	*s2*

In the preceding example, statement *s2* is flow dependent on statement *s1*. The order of these two statements (computations) must be preserved so that the value of A(I) is produced before it is used. Vectorization or parallel execution of this loop does not violate this ordering.

Because Fortran DO loops are just a shorthand notation for repeating similar statements, the original dependence graph is really a compact form of many similar dependence graphs, one for each iteration of the loop. The work tableau in figure 3-1 shows this concept:

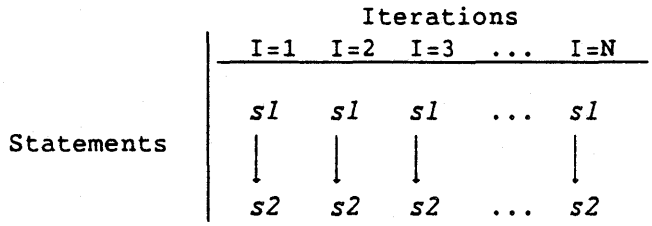


Figure 3-1. Flow Dependence Permitting Vectorization or Multitasking

Figure 3-1 contains no dependence arcs leading from one iteration to another. The loop has *independent iterations*, a property that enables any implementation of the program to execute the iterations in any order and still obtain the same correct answers. In particular, this property enables a multitasked implementation of the program to execute the iterations of the loop simultaneously on multiple processors.

Example:

Code		Dependence Graph
	DO 10 I = 1, N	*s1*
1	A(I+1) = B(I) + 5.	↑
2	B(I+1) = C(I) * 3.	
10	CONTINUE	*s2*

In the preceding example, the dependence graph to the left of the Fortran code shows that statement *s1* is flow-dependent on statement *s2* across iterations of the loop. The value of B(I+1) must be computed in each iteration before it is used in the next iteration; therefore, the loop cannot be vectorized or multitasked unless it is optimized by reordering its statements.

The work tableau for this loop (figure 3-2) shows why the loop does not have independent iterations.

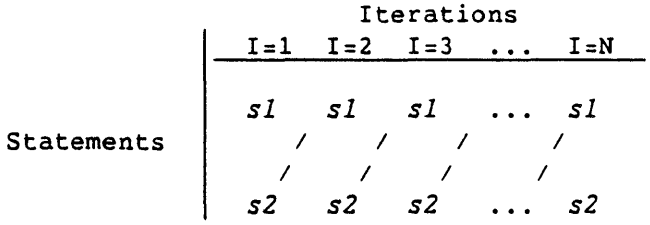


Figure 3-2. Flow Dependence Prohibiting Vectorization or Multitasking

The dependence arcs between iterations prevents the independent execution of the iterations of the loop, because the arcs show that this particular order of execution of the statements must be preserved.

3.2.1.1.2 Antidependence - Statement *s2* is antidependent on statement *s1* if an execution path exists from statement *s1* to statement *s2* and if the same variable (scalar or array element) is used by statement *s1* and assigned by statement *s2*. A directed arc connects node *s1* to node *s2* in the dependence graph.

Example:

Code	Dependence Graph
DO 10 I = 1, N	*s1*
1 A(I) = B(I)	|
2 B(I) = C(I) + 2.	↓
10 CONTINUE	*s2*

In the preceding example, statement *s2* is antidependent on statement *s1*. The order of these two statements (computations) must be preserved so that the value of B(I) is used before it is redefined. Vectorization or parallel execution does not violate this ordering (as shown earlier in figure 3-1).

Example:

Code	Dependence Graph
DO 10 I = 1, N	*s1*
1 A(I) = 2.	↑
2 B(I) = A(I+1) + 1.	|
10 CONTINUE	*s2*

In the preceding example, statement *s1* is antidependent on statement *s2* across the iterations of the loop. The loop cannot be vectorized or multitasked unless the statements are reordered (as figure 3-2 shows).

3.2.1.1.3 Output dependence - Statement *s2* is output-dependent on statement *s1* if an execution path exists from statement *s1* to statement *s2* and if the same variable (scalar or array element) is assigned in both statements. A directed arc connects statement *s1* to statement *s2* in the dependence graph.

Example:

Code	Dependence Graph
DO 10 I = 1, N	*s1*
1 B(I) = C(I)	|
2 B(I) = D(I) * W	↓
10 CONTINUE	*s2*

In the preceding example, statement *s2* is output dependent on statement *s1*. The order of these two statements (computations) must be preserved so that after the loop has completed, B(I) has the value assigned in statement *s2* rather than statement *s1*. Vectorization or parallel execution of this loop does not violate this ordering.

3.2.1.1.4 Unknown dependence - Sometimes the dependence relationship between statements cannot be determined. This can happen in cases such as the following:

- A subscript is subscripted (indirect addressing).
- The subscript does not contain the loop index variable.
- A variable appears more than once with subscripts having different coefficients of the loop variable.
- The subscript is nonlinear in the loop index variable.

When one of these cases occurs, you must assume that a dependence exists, which frequently means you cannot vectorize or multitask the section of code.

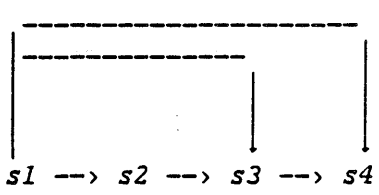
3.2.1.1.5 I/O dependence - A READ statement can be considered equivalent to an assignment statement in which the variable being read is on the left-hand side. Likewise, a WRITE statement can be considered equivalent to an assignment statement in which the variable being written is on the right-hand side. By thinking of READ and WRITE statements in this way, I/O statements can have data dependencies with assignment statements in the same way that assignment statements depend on one another.

Example:

Code

```
      DO 10 I = 1, N
 1      READ *,A(I)
 2      B(I) = A(I)
 3      A(I) = C(I)
 4      WRITE *,A(I)
10    CONTINUE
```

Dependence Graph



In the preceding example, the following dependencies are present:

- *s2* is flow dependent on *s1*.
- *s3* is antidependent on *s2*.
- *s4* is flow dependent on *s3*.
- *s3* is output dependent on *s1*.
- *s4* is flow dependent on *s1*.

Additionally, I/O statements can exhibit a different kind of dependence (I/O dependence) on other I/O statements. This dependence occurs not because the same variable is involved, but because the same file is involved.

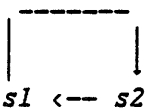
I/O statement *s2* is I/O dependent on I/O statement *s1* if an execution path exists from *s1* to *s2* and if the file referenced in the two I/O statements is the same. A directed arc connects node *s1* to *s2* in the dependence graph. In the preceding example, additional arcs would lead from *s1* to *s1* and from *s4* to *s4* because different iterations access the same files.

Example:

Code

```
      DO 10 I = 1, N
 1      READ(3), A(I)
        REWIND(3)
 2      WRITE(3), B(I)
        REWIND(3)
10    CONTINUE
```

Dependence Graph



In the preceding example, statements *s1* and *s2* are I/O dependent on each other because they both access file 3. Vectorizing I/O is not considered in this example, and it cannot be modified to execute in parallel.

Attempting to make I/O parallel at the loop level can require a significant change to the mechanisms for I/O to preserve the orderliness of the file and correct results. For example, sequential write can be changed to random access, either explicitly through the I/O call or by storing the records into reserved slots in a memory buffer, which is then written when all iterations have completed.

Examples:

The following loops have data-independent iterations (no dependence arcs would exist between iterations in the work tableau, if you were to draw one):

```
      DO 10 I = 1, N
        A(I) = B(I) + C(I)
        D(I) = A(I) + E(I)
        B(I) = F(I)
        B(I) = B(I) + G(I)
 10   CONTINUE


      DO 30 J = 1, N
        DO 20 I = 1, N
          A(I,J) = B(I,J) + C(I,J)
 20     CONTINUE
 30   CONTINUE
```

(The 30 loop and the 20 loop in the preceding example have data-independent iterations.)

```
      DO 40 I = 1, N
        A(I) = SSUM(N,B(1,I),1)
 40   CONTINUE


      DO 60 J = 1, N
        S(J) = FLOAT(J)
        DO 50 I = 1, J
          A(I,J) = FUNCT(S(J),I)
 50     CONTINUE
 60   CONTINUE
      FUNCTION FUNCT(S,I)
      FUNCT = S + FLOAT(I)
      RETURN
      END
```

You cannot determine the data independence of the iterations of either loop in the preceding example without analyzing the FUNCT subprogram. The iterations of both loops are independent because the subroutine does not redefine its input (no side effects).

The following loops have data-dependent iterations (dependence arcs would be present between iterations in a work tableau):

```
      S = 0.0
      DO 10 I = 1, N
        S = S + A(I)
10    CONTINUE


      DO 20 I = 1, N
        X(I) = A(I)*X(I-1) + B(I)
20    CONTINUE


      DO 40 J = 1, N
        A(J) = 0.0
        DO 30 I = 1, N
          A(J) = A(J)*C(I) + B(I,J)
30      CONTINUE
40    CONTINUE
```

(While the 30 loop has data-dependent iterations, the 40 loop has data-independent iterations.)

```
      DO 60 J = 1, N
        S(J) = FLOAT(J)
        DO 50 I = 1, J
          A(I,J) = FUNCT(S(J),I)
50      CONTINUE
60    CONTINUE
      FUNCTION FUNCT(S,I)
      S = SQRT(S)
      FUNCT = S + FLOAT(I)
      RETURN
      END
```

The 60 loop has data-independent iterations, but the 50 loop does not.

3.2.1.2 Control dependence

Control dependence refers to the situation in which the order of execution of statements cannot be determined before run time. Such is the case when conditional statements (for example, IF) appear in a program. Conditional statements conditionally introduce or conditionally eliminate data dependence among statements.

Because computational independence must be guaranteed when you attempt multitasking, a cautious approach is necessary. You must assume that any conditional statement may or may not cause the execution of its object statement. Analyze all possible execution paths through the code to be multitasked, ensuring that no data dependencies are eliminated.

If an IF statement tests a value computed earlier in the same iteration, the testing does not introduce a data dependence among iterations. On the other hand, if an IF statement tests a value computed in another iteration, the iterations are dependent and cannot be done in parallel.

If the object statement of an IF statement assigns a variable used later in the same iteration, the assignment does not introduce data dependence among iterations. If, however, the object statement of an IF statement assigns a variable used in another iteration, the iterations are dependent and cannot be executed in parallel.

Examples:

The following loops have control-independent iterations:

```
      DO 10 I = 1, N
        A(I) = B(I)
        IF( A(I).LT.0.0 ) A(I) = 0.0
 10   CONTINUE


      DO 20 I = 1, N
        IF( I.LT.10 ) GO TO 15
        IF( I.GT.20 ) GO TO 16
 15     A(I) = B(I)
        GO TO 20
 16     A(I) = C(I)
 20   CONTINUE


      DO 30 J = 1, N
        IF( A(J).EQ.B(J) ) GO TO 26
        DO 25 I = 1, J
          IF( C(I).EQ.A(J) ) GO TO 25
          D(I,J) = A(J)
 25     CONTINUE
 26     CONTINUE
 30   CONTINUE
```

The following loops have control-dependent iterations:

```
      DO 40 I = 1, N
        IF( A(I-1).EQ.0.0 ) A(I) = 1.0
 40   CONTINUE


      K = J
      DO 50 I = 1, N
        A(I) = 1.0
        IF( B(I).EQ.0.0 ) A(K) = 0.0
        K = K + 1
 50   CONTINUE


      DO 70 J = 1, N
        DO 60 I = 1, N
          S(J) = AMAX(S(J),X(I))
 60   CONTINUE
 70   CONTINUE
```

In the last example, while the 60 loop has control-dependent iterations, the 70 loop has control-independent iterations.

3.2.2 STORAGE DEPENDENCE

While computational dependence is concerned with the independence of the work to be done, *storage dependence* is concerned with the independence of workspace. Each parallel computational task has access to variables, and the fetching and storing of all variables in one task must not interfere with that in another task. Each task must work on independent storage locations or use special access mechanisms (for example, locks) to guarantee the safe modification of shared variables.

Storage dependence is a data dependence between the iterations of a loop caused by the extent of a left-hand side variable being less than the index range of the loop considered for multitasking. The same storage location is reused and redefined in each iteration, causing a cycle in the dependence graph for the loop and introducing dependence arcs between iterations in the work tableau.

Modifying the program to provide each iteration (or the partition of iterations belonging to each task) with a separated storage location corresponding to the original variable can frequently eliminate data dependencies between loop iterations.

Storage dependence is easily overlooked and often difficult to identify as a multitasked program bug.

Examples:

The following loops have storage-independent iterations:

```
      DO 10 I = 1, N
        A(I) = B(I) + C(I)
  10  CONTINUE


      DO 20 I = 1, N
        IF( A(I).EQ.0.0 ) GO TO 20
        B(I) = C(I)/A(I)
  20  CONTINUE


      DO 30 J = 1, N
        DO 29 I = 1, N
          A(I,J) = 0.0
          DO 28 K = 1, N
            A(I,J) = A(I,J) + B(I,K)*C(K,J)
  28      CONTINUE
  29    CONTINUE
  30  CONTINUE
```

In the last example, the 30 and 29 loops have storage-independent iterations, but the 28 loop does not. The variable A(I,J) has an extent that is 1 over the range of K.

The following loops have storage-dependent iterations:

```
      S = 0.0
      DO 40 I = 1, N
        S = S + A(I)
  40  CONTINUE


      DO 70 K = 1, N
        DO 50 J = 1, N
          A(J) = B(J,K) + C(J,K)
  50    CONTINUE
        DO 60 I = 1, N
          D(I,K) = A(I) + 1.0
  60    CONTINUE
  70  CONTINUE
```

In this example, the 50 and 60 loops have storage-independent iterations, but the 70 loop does not. The variable A(J) has an extent that is 1 over the range of K.

3.2.3 GENERALIZATIONS

Applying the concepts of statement-level dependence at a higher level, such as the code segment, subroutine, process, or task level, is frequently useful. You can infer the dependence of two higher-level objects from knowing which statements of one object depend upon the statements of the other object, and how.

The goal of analyzing the dependencies in a code is to identify the variables and program constructs that need modification before you can multitask the code. Section 9, Advanced Macrotasking in Fortran, contains more discussion on analysis, and section 6, Program Analysis and Conversion, describes conversion techniques.

3.3 SCOPE

The scope of a variable is the region of the program in which the variable is defined and can be referenced. The traditional regions in which a variable has scope include statement, program unit, and executable program. Outside of a variable's scope, the variable is not defined and references to the variable's name either refer to a different variable and distinct memory location or result in undefined variable errors.

The boundaries of a variable's scope are easy to recognize. They may be the beginning and end of a statement, the first and last line in a program unit, or a collection of program units sharing the variable using a COMMON block or argument list mechanism.

When a program is partitioned into pieces to be executed by several CPUs, a set of independent iterations of a loop is formed into a subroutine task. The effect of this partitioning is to introduce subroutine scope boundaries in the program where DO and CONTINUE statements occur. The program can be partitioned at any DO-loop level, provided that the loop has independent iterations.

A basic step in the process of converting a code for multitasking is to understand the scope of the variables in the original program and what the new scope of variables should be in the multitasked code. It is necessary to analyze the ramifications of introducing a new scope boundary and to provide for the sharing or privacy of variables according to their multitasking use (see subsection 9.5, Shared and Private Variables). Figure 3-3 shows the alteration of scope boundaries in dividing a piece of work for multitasking.

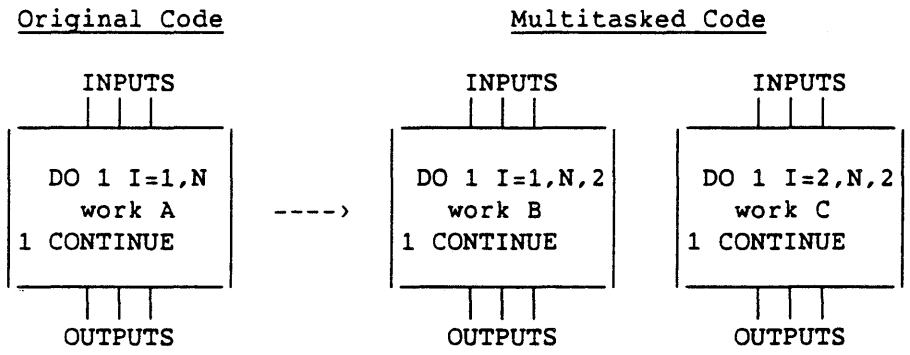


Figure 3-3. Changes in Scope Boundaries when Multitasking a Code Segment

Figure 3-3 shows work A considered for multitasking by dividing it into parts B and C, each of which is executed by independent tasks. The variables in work A have relationships (scope and dependencies) with the inputs and outputs to work A. You must accommodate these relationships in the multitasked code. For example, the inputs to A may need to be available to the task that executes work C, and the outputs of work C may need to be available to the code that follows work A. In addition, the tasks that execute work B and C may need individual local (private) copies of single variables that appear in work A.

The variables in the original program can be categorized according to their scope. The scope is defined with respect to the loop to be multitasked. Your scope analysis of the original program must consider all variables referenced in the loop and must consider the possible references to these variables made outside the loop.

The following are scope categories for variables in programs that have not been multitasked:

- Variables local to the loop
- Variables whose scope extends beyond the loop
    - But are local to the subroutine containing the loop
    - And extends beyond the subroutine by argument list
    - And extends beyond the subroutine by COMMON block

The following example shows variables having these different scopes:

```
      SUBROUTINE SUB ( A )
      DIMENSION A(100)
      COMMON / B / B(100)
      C = 0.
      DO 10 I = 1, 100
        D = A(I) + 1.
        A(I) = B(3) - D
        C = C + B(3)*D
   10 CONTINUE
      PRINT, C
      RETURN
      END
```

In the preceding example, variable D is local to the loop, not being used outside the loop or cumulatively altered over the iterations of the loop. Variable C is global to the loop, being referenced outside the loop and being cumulatively changed over successive iterations, though it is local to the subroutine SUB. The scope of A extends beyond the loop by an argument list, and the scope of B extends beyond the subroutine by a COMMON block mechanism.

3.4 DETERMINISM

The execution order for statements in programs that run on only one CPU is well defined. Repetitive runs produce identical results because the same instructions are executed in the same order each time. Data and control dependencies are satisfied simply by the location of statements in the program; the order of execution is determined by the order of statements within the program.

Multitasking introduces a new dimension to the order of execution. While the sequence of execution in each task remains well defined, the relative order of task execution has no default order. In fact, the order of execution may change from run to run, and you must control the ordering to satisfy dependencies. Failure to manage the temporal ordering of tasks is a subtle error that can be difficult to identify.

Consider the following program segment:

```
      DO 10 I = 1, N
        A(I) = AINIT(I)
 10   CONTINUE
      DO 30 J = 1, N
        DO 20 I = 1, N
          B(I,J) = A(I)*C(J)
 20     CONTINUE
 30   CONTINUE
```

This segment is incorrectly converted to the following:

```
      CALL TSKSTART( IDTASK, T, N, A, AINIT, B, C )
      DO 10 I = 1, N/2
        A(I) = AINIT(I)
 10   CONTINUE
      DO 30 J = 1, N/2
        DO 20 I = 1, N
          B(I,J) = A(I)*C(J)
 20     CONTINUE
 30   CONTINUE
      CALL TSKWAIT( IDTASK )

      SUBROUTINE T ( N, A, AINIT, B, C )
      DO 10 I = N/2+1, N
        A(I) = AINIT(I)
 10   CONTINUE
      DO 30 J = N/2+1, N
        DO 20 I = 1, N
          B(I,J) = A(I)*C(J)
 20     CONTINUE
 30   CONTINUE
      RETURN
      END
```

The resulting code may not produce the right answers, because both 10 loops may not finish before either 20 loop begins. Synchronization is required between the 10 and 30 loops in both tasks to ensure compliance with the data dependencies of A. The following is a correct program:

```
      COMMON / EVENTS / IDONE1, IDONE2
      CALL EVASGN(IDONE1)
      CALL EVASGN(IDONE2)
      CALL TSKSTART( IDTASK, T, N, A, AINIT, B, C )
      DO 10 I = 1, N/2
        A(I) = AINIT(I)
10    CONTINUE
      CALL EVPOST( IDONE1 )
      CALL EVWAIT( IDONE2 )
      CALL EVCLEAR( IDONE2 )
      DO 30 J = 1, N/2
        DO 20 I = 1, N
          B(I,J) = A(I)*C(J)
20      CONTINUE
30    CONTINUE
      CALL TSKWAIT( IDTASK )

      SUBROUTINE T ( N, A, AINIT, B, C )
      COMMON / EVENTS / IDONE1, IDONE2
      DO 10 I = N/2+1, N
        A(I) = AINIT(I)
10    CONTINUE
      CALL EVWAIT( IDONE1 )
      CALL EVCLEAR( IDONE1 )
      CALL EVPOST( IDONE2 )
      DO 30 J = N/2+1, N
        DO 20 I = 1, N
          B(I,J) = A(I)*C(J)
20      CONTINUE
30    CONTINUE
      RETURN
      END
```

3.5 SPEEDUP FROM MULTITASKING

Multitasking produces the best speedup when applied to balanced tasks of sufficient size. Speedup occurs only when multiple processors have something to do and when the time saved in executing independent tasks in parallel outweighs the overhead penalty.

3.5.1 TASK GRANULARITY

The initiation, management, and interaction of tasks is accomplished by added code that increases the CPU time of the program and limits the granularity of parallelism that can be profitably exploited.

When converting a program for multitasking, look at the size of the task to see whether multitasking will produce a speedup. The following program can represent potential extremes in multitasking:

Original code:

```
      PROGRAM MAIN
      CALL A              Subroutines A and B perform
      CALL B              independent operations
      STOP                on independent data.
      END
```

Multitasked code:

```
      CPU-0                     CPU-1

      PROGRAM MAIN
      CALL TSKSTART(TID,A) --> SUBROUTINE A
      CALL B
      CALL TSKWAIT(TID) <----- RETURN
      STOP
      END
```

This program can benefit from multitasking, depending on the size (in execution time) of subroutines A and B, the difference in size between A and B, and the multitasking overhead. Table 3-1 shows tasks of varying granularity. It specifies the task time in both seconds and clock periods (CPs). Parallelism exists in each of the tasks shown, with increasing granularity of parallelism toward the top. The size of these examples is a function of the number of operations, N. For certain values of N, some of these examples may profit from multitasking, while others are too small to consider. Consider the following matrix addition example:

Original code:

```
      DO 1  J = 1, N
        DO 1 I = 1, N
          A(I,J) = B(I,J) + C(I,J)
1     CONTINUE
```

Multitasked code (static load balancing):

```
        CPU-0                                CPU-1

      COMMON /GLOBAL/ A,B,C
 A-->
      L = N/2
      LP1= L + 1
 B-->
      CALL TSKSTART(IDT,T,LP1,N) -->
 C-->                                                    <--F
                                     SUBROUTINE T(LP1,N) <--G
                                     COMMON/GLOBAL/A,B,C
      DO 1  J = 1, L                 DO 1  J = LP1, N
        DO 1  I = 1, N                 DO 1  I = 1, N
          A(I,J)=B(I,J)+C(I,J)           A(I,J)=B(I,J)+C(I,J)
    1 CONTINUE                     1 CONTINUE
                                     RETURN              <--H
                                     END
 D-->
      CALL TSKWAIT(IDT) <-----------
 E-->
```

Table 3-1. Sample Tasks Containing Parallelism

Task Time		Number of Operations	Representative Task
Seconds	CPs		
10**(-2)	10**6	N^4	DO 4 I = 1, N 4 CALL MATMUL(N,A,B,C)
10**(-3)	10**5	N^3	DO 3 I = 1, N DO 3 J = 1, N A(I,J) = 0. DO 3 K = 1, N 3 A(I,J) = A(I,J) + B(I,K)*C(K,J)
10**(-4)	10**4	N^2	DO 2 J = 1, N DO 2 I = 1, N 2 A(I,J) = B(I,J) + C(I,J)
10**(-5)	10**3	N	DO 1 I = 1, N 1 V(I) = U(I) + W(I)
10**(-6)	10**2	1	S = X1 + X2 + X3 + X4

Significant events in the execution of the multitasked code are as follows (the event labels are from the preceding example):

Event Label	Event
A	The experiment begins. CPU-0 executes setup code.
B	CPU-0 calls TSKSTART.
C	CPU-0 resumes its own processing.
D	CPU-0's half of work complete; CPU-0 calls TSKWAIT.
E	The experiment ends, synchronization occurs, and CPU-0 continues.
F	CPU-1 becomes aware of the new task.
G	CPU-1 begins processing its half of the work.
H	CPU-1 completes the work; the task dies.

Figure 3-4 shows these events on a time line.

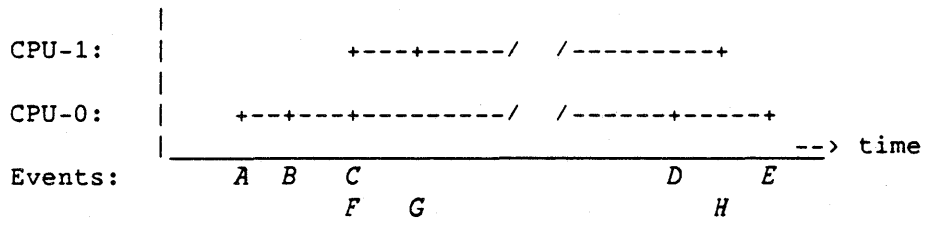


Figure 3-4. Time Line for a Two-CPU Multitasking Example

The following assumptions and observations can be made on the basis of this time line:

- The execution time for one CPU would be 2 * (D-C) = X.
- The execution time for two CPUs is E - A.
- The time at event C is equal to the time at event F.
- The multitasking overhead is (C-A) + (E-D) for CPU-0 and (G-F) for CPU-1.
- The speedup factor for multitasking is as follows:

$$\text{Speedup} = \frac{2 * (D-C)}{(E-D)+(D-C)+(C-A)} = \frac{\text{Time(one CPU)}}{\text{Time(two CPUs)}}$$

- The following are estimates for the execution times of various program segments (the segment boundaries are indicated by the event labels from the previous example of static load balancing):

  (*B–A*) = 50 CP
  (*C–B*) = 4000 CP
  (*D–C*) = Independent variable X/2 = 1/2 of one-CPU execution time
  (*G–F*) = 100 CP
  (*H–D*) = (*G–F*)
  (*H–G*) = (*D–C*)
  (*E–H*) = 1000 CP
  C = *F*

- For this multitasking model, the overhead is simply the sum of the individual overheads for CPU-0, because the overhead to CPU-1 eventually causes CPU-0 to wait. The expected speedup is a function of the size of the original task.

$$Sp = \text{Speedup} = \frac{X}{X/2 + 5150} \qquad \text{Units(X)} = \text{CP}$$

Figure 3-5 plots the speedup function for matrix addition to show when the multitasked code will be slower than the original code, the same speed, and faster.

The original task execution time is related to the order of the matrix sum, N, and the computation's mode: scalar or vector. The important measurement of task size should be made in terms of execution time, not floating-point operations, because the number of operations that can be executed in a given amount of time depends greatly on the degree of vectorization.

The following calculations determine the task granularity for matrix addition:

```
Speedup = 1.0  <-->  Task size = 10300 CP = O(10**(-4)Sec)
                              N = 50 (vector),  N = 15 (scalar)

Speedup = 1.8  <-->  Task size = 92700 CP = O(10**(-3)Sec)
                              N = 200 (vector), N = 50 (scalar)
```

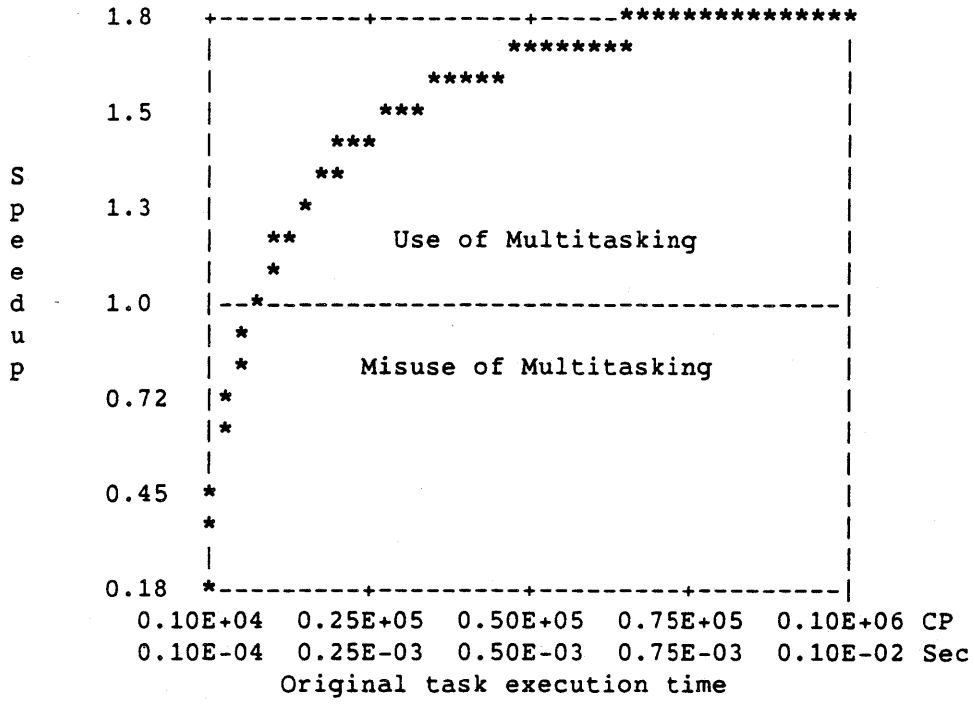


Figure 3-5. Speedup of a Multitasked Matrix Addition

For simple multitasking models, the following formula gives the size of the original task required to obtain a desired speedup. The formula does not address portions of the program that are not multitasked and should not be applied to the program as a whole.

$$X = \frac{Sp * ncpu * overhead}{(ncpu - Sp)}$$

To gain a speedup of *Sp* on *ncpu* processors, with a multitasking overhead of *overhead* CPs, the original task must have an execution time of at least X CPs.

3.5.2 LOAD BALANCING

To make the best use of processor resources and gain the most speedup, you must partition the work into equal parts to be run parallel. A comparison of figures 3-6 and 3-7 shows why this is important.

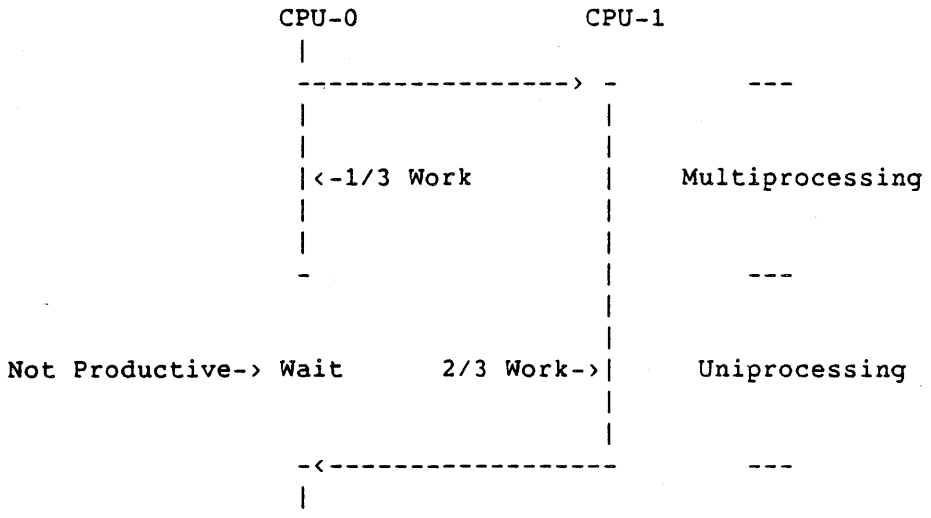


Figure 3-6. An Unbalanced Multitasked Job

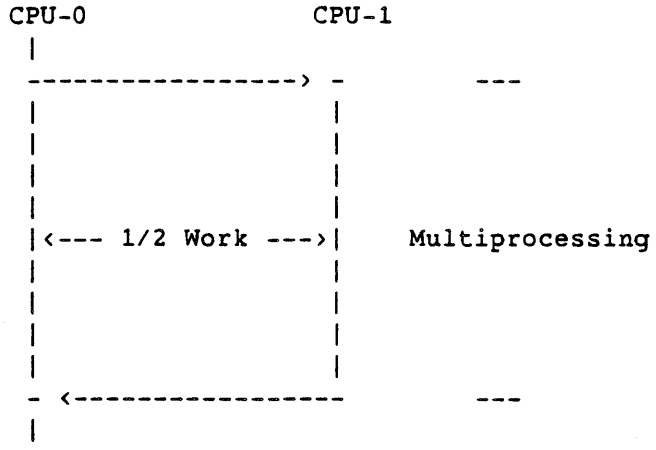


Figure 3-7. A Balanced Multitasked Job

Multitasking is most often applied to parallel work in the independent iterations of DO loops. If the loop has N iterations, N is called the extent of the parallelism of the loop. Load balancing is the technique of mapping N onto P processors or tasks so that each task has the same amount of work to do (see figure 3-8).

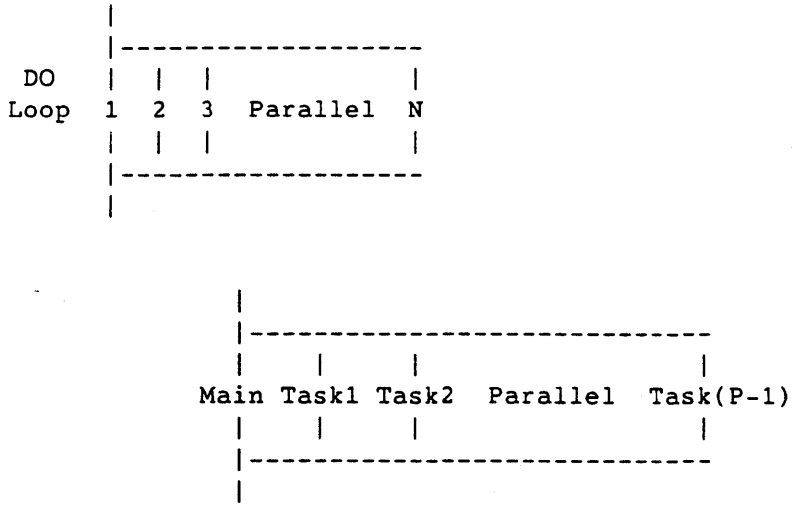


Figure 3-8. Multitasked Code with N Iterations on P Processors

In load balancing, two major cases must be considered: static and dynamic partitioning. Use static partitioning when the times for each of the loop iterations are approximately equal. This technique assigns a fixed subset of iterations to each task. The iterations a task is to perform can be computed from the task or processor number. One strategy assigns a subset of contiguous iterations to each processor. The *n*th processor computes iterations (*n**N)/P+1 to ((*n*+1)*N)/P.

Example of contiguous static partitioning with P = 4:

Processor	Assigned Iterations
P0	I = 1, N/4
P1	I = N/4+1, 2*N/4
P2	I = 2*N/4+1, 3*N/4
P3	I = 3*N/4+1, N

Take care in computing loop bounds to ensure that no iterations are missed or duplicated.

Another static strategy assigns the iterations in an interleaved fashion. The *n*th processor computes every Pth iteration from *n*+1 to N.

Example of interleaved static partitioning with P = 4:

Processor	Assigned Iterations
P0	I = 1, N, P
P1	I = 2, N, P
P2	I = 3, N, P
P3	I = 4, N, P

If the times for iterations of the loop vary significantly, dynamically partitioning the work tends to balance the load on each processor. This technique maintains a shared counter, which indicates the next unprocessed iteration. Each processor accesses and updates the counter to commit itself to one or more iterations. The processors that commit to short iterations return more often to find more work, while processors committed to long iterations look for new work less frequently. This technique incurs an overhead to protect the counter during the update process. If the average granularity of the iterations is large compared to this overhead, the iterations may be distributed one at a time. Let the shared counter be I, set initially to 1; then each processor can execute the code in the following example:

Example of dynamic partitioning for large-granularity iterations:

```
  10  CONTINUE
      CALL LOCKON ( LOCKI )
      L = I
      I = L + 1
      CALL LOCKOFF ( LOCKI )
      IF ( L.GT.N ) GO TO 20
C        compute iteration L
      GO TO 10
  20  CONTINUE
```

If the average granularity of the iterations is small compared to the overhead of protecting the iteration counter, the iterations can be distributed to processors in chunks. Each chunk contains K iterations. The code is similar to the previous code.

Example of dynamic partitioning for small-granularity iterations:

```
  10  CONTINUE
      CALL LOCKON ( LOCKI )
      L = I
      I = L + K
      CALL LOCKOFF ( LOCKI )
      IF ( L.GT.N ) GO TO 20
C        compute iterations L through min(L+K-1,N)
      GO TO 10
  20  CONTINUE
```

Take care when programming such an infinite loop. Choose the value of K for its effects on overhead and load balancing; too small a value of K does not allow the lock overhead cost to be recovered over enough iterations, and too large a value of K does not produce enough chunks to allow load balancing to occur. Figure 3-9 shows these trade-offs.

To reduce the relative overhead, K should be at least K_{min}.

$$K_{min} = \text{smallest K such that } \frac{\text{(lock overhead)}}{\text{K*(average iteration time)}} < .01$$

To facilitate load balancing, K should be less than K_{max}.

$$K_{max} = \text{largest K such that } \frac{N}{K} > 10*P$$

Choose the value of K in the range (K_{min}, K_{max}).

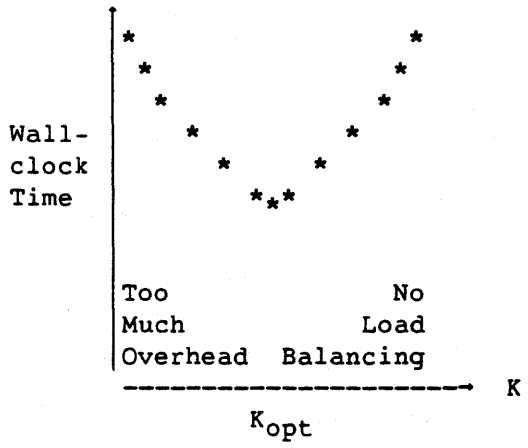


Figure 3-9. Trade-offs in Selecting a Chunking Factor (K)

Example of dynamic partitioning with microtasking:

```
CMIC$ DO GLOBAL BY K
      DO 20 L = I, N
C     Compute an iteration
 20   CONTINUE
```

3.6 PREDICTING PERFORMANCE

Predicting the performance of multitasked code lets you see the benefits of multitasking before you begin converting.

3.6.1 FACTORS AFFECTING PERFORMANCE

Several factors influence the performance of multitasked code as compared to that of the original program. Some of these factors are changes required for multitasking to occur, including reentrant code generation to produce stack-based references for local variables.

All tasks contend for access to shared Central Memory for their code and variables.

While the preceding factors are largely out of your control, you can influence the following factors, which frequently have a greater impact on performance:

- Level (granularity) of parallelism exploited
- Frequency of calls to the multitasking library
- Partitioning and distributing work among processors
- Programming style in the choice of multitasking mechanisms

Of all the factors listed, the most important are granularity of task and balanced workload distribution.

3.6.2 MANUAL PERFORMANCE PREDICTION

This subsection analyzes a simple program to predict its performance if it is converted for multitasking. The original code has the following structure:

```
      PROGRAM MAIN
      ...
      DO 100 I = 1, 50
      ...
        DO 10 J = 1, 2
          CALL SUB(J)
 10     CONTINUE
      ...
100   CONTINUE
      ...
      STOP
      END
```

Analyzing the program indicates that the 100 loop has dependent iterations and cannot be executed in parallel. The 10 loop has independent iterations and multitasking can be attempted at this level. Execution on one CPU shows that 96% of the execution time is spent in subroutine SUB with an average time of 0.2 seconds per call. The total run time is 20.83 seconds.

First, compute the theoretical speedup, assuming no multitasking overhead or other performance degradation.

```
Execution time (one CPU) = Time(Seq) + Time(Mt)
                         = (4% + 96%) * 20.83 seconds
                         = 100% * 20.83 seconds
                         = 20.83 seconds

Execution time (two CPU) = Time(Seq) + (1/2) * Time(Mt)
                         = (4% + 48%) * 20.83 seconds
                         = 52% * 20.83 seconds
                         = 10.83 seconds
```

The maximum attainable speedup is calculated as follows:

$$\text{Speedup} = \frac{\text{Time (one CPU)}}{\text{Time (two CPUs)}} = \frac{20.83}{10.83} = 1.92$$

This is a measure of the parallelism exploited in the program.

Next, conceptually convert the program to the following multitasking structure (not considering details of scope and storage independence):

```
      PROGRAM MAIN
      COMMON/MT/ISTART,IDONE,JOB
      CALL TSKSTART(IDTASK,T) ----> SUBROUTINE T
      JOB = 1                       COMMON/MT/ISTART,IDONE,JOB
      DO 100 I = 1, 50           1  CALL EVWAIT(ISTART)
        CALL EVPOST(ISTART) --->    CALL EVCLEAR(ISTART)
        CALL SUB(1)                 IF( JOB.NE.1 ) GO TO 2
        CALL EVWAIT(IDONE)          CALL SUB(2)
        CALL EVCLEAR(IDONE) <---    CALL EVPOST(IDONE)
 100  CONTINUE                      GO TO 1
      JOB = 2                     2 RETURN
      CALL EVPOST(ISTART) <------   END
      CALL TSKWAIT(IDTASK)
      STOP
      END
```

You can now project a realistic two-CPU execution time from the intrinsic costs of the calls to the multitasking library and by estimating other relevant quantities.

```
Execution time (two CPUs) = Time(Seq) + (1/2) * Time(Mt) + Overhead
```

where

```
Overhead =       Time(TSKSTART)   +
                 Time(TSKWAIT)    +
               51*Time(EVPOST)    +
               50*Time(EVCLEAR)   +
               50*Time(EVWAIT)    +
        Workload imbalance delay  +
          Memory contention delay
```

Compute the overhead in terms of the main task. Overheads in task T either are masked by delays in MAIN or cause delays in T that are accounted for in the workload imbalance or memory contention delays in MAIN. Execution times for the calls to the multitasking library are available in appendix C, Approximate Timings.

Approximate the delays caused by workload imbalance and memory contention. This example assumes no workload imbalance and estimates the memory contention at 2%. This percentage is applied to all the time spent in SUB when it is called from MAIN, the time when both tasks are accessing Central Memory. With these estimates, you can compute the overhead.

```
Overhead =        1500000 CP    +
                     1500 CP    +
                51*( 1500 CP )  +
                50*(  200 CP )  +
                50*( 1500 CP )  +
                       0        +
        (0.02)*50*( 0.2 seconds)
```

This calculation gives an overhead of 0.216 seconds, which you can use in projecting a realistic two-CPU execution time.

```
Execution time(two CPUs) = Time(Seq) + (1/2) * Time(Mt) + Overhead
                         = (0.83 + 10.0 + 0.216) seconds
                         = 11.046 seconds
```

You can use this estimate of execution time to compute a realistic speed-up projection.

$$\text{Speedup} = \frac{\text{Time(one CPU)}}{\text{Time(two CPUs)}} = \frac{20.83}{11.046} = 1.88$$

Finally, from this projected speedup, you can determine whether multitasking is worthwhile, weighing the gain against the required conversion effort.

The actual speedup recorded for this example was 1.88.

3.7 CHOOSING VECTORIZATION OVER MULTITASKING

Multitasking offers a speedup, $Sp < ncpu$, over conventional processing, enhancing either scalar or vector performance. Because vector processing offers a greater speedup potential over scalar processing than does multitasking, multitasking should not be employed at the expense of vectorization. In the case of a short vector length, scalar processing outperforms vector processing. Similarly, in the case of a small task size, vector processing (or even scalar processing) may outperform multitasking.

Consider the following simple loop:

```
      DO 10 I = 1, N
        A(I) = B(I) + S*C(I)
 10   CONTINUE
```

Depending on the value of N, the best performance may come from executing this loop in scalar mode, vector mode, or multitasked vector mode. If N is very small, scalar processing may be best, while if N is large, multitasking is appropriate. The speedup depends on the overhead of the particular multitasking mechanism used (see section 4, Microtasking).

4. MICROTASKING

Microtasking permits multiple processors to work on a Fortran program at the DO-loop level. This section shows the experienced Fortran programmer how to analyze a code to determine what may be microtasked, how to insert the required directives, how to avoid typical beginners' errors, and how to measure improvements in performance.

Cray Fortran users are already accustomed to speeding up their programs by using vector hardware to perform operations on inner DO loops that have no data dependencies. In fact, most such vectorization is done automatically by the CFT and CFT77 compilers. The compiler also points to data dependencies that inhibit vectorization. The programmer can revise the code or issue compiler directives (of the form CDIR$...) to further enhance performance.

Once such optimizing is complete, a single processor can work no faster, but more than one processor could operate on separate parts of the data simultaneously to achieve results faster. This is what microtasking accomplishes. The name *microtasking* was chosen because multiple processing is efficient even at a DO-loop level where the task size, or granularity, may be small.

In addition to working efficiently on parts of programs where the granularity is small, microtasking works well when the number of processors available for the job is unknown or may vary during the program's execution. Microtasking overhead is generally much smaller than macrotasking overhead. The overhead of synchronization is small enough that when only one processor is available, microtasking adds only a small increase in execution time over the version without microtasking. Additionally, in a batch environment where processors may become available from time to time for very short periods, the microtasked job can dynamically adjust to the number of available processors.

This means that microtasked jobs do not require a dedicated system, though, as with most programs, they perform best in a dedicated environment with no competing jobs. The most efficient use of a multiprocessor system in a batch environment is to assign each processor to a separate job. This cannot always be done.

When fewer jobs fit in memory than the total number of processors, one or more processors are forced to be idle; with microtasking, these idle processors can be assigned to work on different portions of one job. For example, an 8-million-word, four-processor Cray computer system has three 2½-million-word jobs and no other jobs small enough to use the remaining memory. One of those jobs, if microtasked, can use the extra processor until a job comes in that is small enough to use the remaining memory. When such a small job does enter the system, the microtasked job and the new small job will share the CPU. This will more efficiently use processors and will improve the system's overall performance.

The small granularity permitted by microtasking also simplifies the programmer's job; less analysis is required to find data dependencies than is necessary when multitasking larger pieces of code. This is not to say that microtasking cannot be done at a very high level if the scope of data permits it; it simply is not required for efficient operation.

Also, the programmer need not rewrite substantial portions of the code. A preprocessor, PREMULT, converts the user's Fortran by interpreting a small number of directives and then rewriting the program to make microtasking library calls. The original code is still standard Fortran, because the directives are treated as comment lines. The resulting code is a mixture of Fortran and CAL, incorporating calls to library routines specific to CRI.

Microtasking and the macrotasking routines (see section 5, Macrotasking) complement each other. Macrotasking routines work better for systems in which the parallel work is disjointed and each processor can work independently on its own piece of work without much synchronization with other processors. Microtasking works better when the task size is small, when parallel tasks are tightly coupled, and when dedicated time is not available. You can use both microtasking and macrotasking in the same program.

4.1 MICROTASKING TERMS AND CONCEPTS

From the user's point of view, a microtasked job has sections containing data that can be operated on in parallel without adversely affecting other data. The objective of microtasking is to let the computer process those sections in parallel. The major consideration that limits parallelism within a program is the scope of data items. Your first job, then, is to understand the data in the program. It is also important to forget preconceptions you may have about needing to keep track of processors and what each is doing at any time. Concentrate instead on the data: where it is accessed and what is done with it. For this reason knowing where to use microtasking is a data problem, not a coding problem. A matrix multiplication program example follows:

```
      SUBROUTINE MXM(A,B,C,L,M,N)
      DIMENSION A(L,M),B(M,N),C(L,N)
C
      DO 100  K = 1,N
        J = 1
        DO 200 I = 1,L
          C(I,K) = A(I,J)*B(J,K)
  200   CONTINUE
C
      DO 100 J = 2,M
        DO 100 I = 1,L
          C(I,K) = C(I,K) + A(I,J) * B(J,K)
  100 CONTINUE
      RETURN
      END
```

Two matrices are passed into the subroutine and a result matrix is passed out. Only the result matrix changes as a result of calling the subroutine. The matrices passed in and out of the subroutine are called *global* data. You must understand how changes are made to global data before microtasking a subroutine, because those changes determine the microtasking structure you will need to impose on the code. Identifying global data is explained in detail later in this section.

Inside the subroutine, every element of the result matrix is computed as the dot product of the relevant row and column of the input matrices. Every result is independent of every other result. On a vector processor, it is efficient to group operations to take advantage of the vector hardware, but it does not matter whether, for example, column 8 is done before or after column 676. On a multiprocessing machine, each processor can work on one column at a time without regard to the number of processors doing the work or the number of columns each works on.

In fact, this willingness to let activity in a subroutine be this disordered led to calling a microtasked subroutine a *fray*. One column is the smallest portion of data you want worked on at one time, and that piece of work is called a *process*. Work on a column can be done at the same time as work on the other columns, and these multiple processes are grouped into a *control structure* to ensure that all are completed. A microtasked version of the subroutine for matrix multiplication follows:

```
      SUBROUTINE MXM(A,B,C,L,M,N)
      DIMENSION A(L,M),B(M,N),C(L,N)
C
CMIC$ DO GLOBAL
      DO 100  K = 1,N
        J = 1
          DO 200 I = 1,L
            C(I,K) = A(I,J)*B(J,K)
  200     CONTINUE
C
      DO 300 J = 2,M
        DO 300 I = 1,L
          C(I,K) = C(I,K) + A(I,J) * B(J,K)
  300 CONTINUE
  100 CONTINUE
      RETURN
      END
```

The directives are explained later, but the example shows that the work on any one column is independent of the work on any other column and may therefore proceed in parallel with the work on other columns. All changes to global data are made inside the area marked off by the CMIC$ DO GLOBAL directive, which ends with the end of the DO loop. This is the control structure. The separate iterations of the outermost DO loop are the processes.

A second example, microtasking a Fast Fourier Transform (FFT), shows how to structure a subroutine in which items in the result array depend on values of other items in the result array. This example shows how to give each processor that works in the subroutine its own unique array as a scratch work space.

```
      SUBROUTINE FFT2D(A,N,M)
      DIMENSION A(N,M),WORK(2048)
      CALL FFTINIT(N,WORK)
      DO 100 I=1,M
        CALL FFTCOL(A(1,I),N,WORK)
  100 CONTINUE
      CALL FFTINIT(M,WORK)
      DO 200 I=1,N
        CALL FFTROW(A(I,1),M,WORK)
  200 CONTINUE
      RETURN
      END
```

This two-dimensional FFT requires two control structures. First, each column of length N constitutes a process that can be calculated independently. However, all columns must be processed before any row can be started. Thus, each row of length M is also a process, but the row calculations taken together constitute a separate control structure because they cannot be performed in parallel with the column calculations.

The work of FFTINIT must, in each case, be done by each processor that does work in the subroutine, because the WORK array contains both constants needed for the calculations and scratch space used for each iteration of the DO loops. The array WORK is local data, which means that each processor has its own copy of WORK, and the final results of WORK are not used in the results of the calling program. It is conventional to mix the constants and scratch space in this way in FFTs. Each processor must have its own scratch space; otherwise, it may overwrite space being used by another processor. The constants take very little time to set up compared to the overall time spent in the subroutine. In fact, to set them up once for all processors and then keep track of them during the multiple processing requires extra work if the conventional storage scheme is maintained. Thus, it is more effective to have each processor set up its own copy of the same constants.

The array A and dimensions M and N are global data. The results that are stored into A are used in the results of the calling program. You must ensure that within a control structure no element of A is updated by more than one processor. Processing the rows and columns separately does this. M and N represent global constants and are presumably not updated at all; you must ensure that this is the case, or an error can result. The value I is local, and a different value is passed to each processor that takes a column or a row of work to do.

Any available processor arrives at a control structure and takes a piece of work. The processor must complete the piece of work it has taken, but it is not thereby committed to accepting any more work. When the second DO loop is completed, a single processor returns to the calling routine. Note especially that it is permissible, even likely, that a processor that has not done any work in the first control structure will arrive in time to accept work in the second. It simply bypasses the first control structure and enters the second.

In the FFT example, you need not be concerned with the size of the problem or the number of processors available to work on the job. You should be concerned exclusively with where local data is modified and where global data is modified. This distinction determines what constitutes a process and what constitutes a control structure.

The previous example is typical of most microtasking in that most of the work of a microtasked routine occurs within control structures. The average program run on Cray computer systems exists primarily to modify data in large arrays, those that would be operated on in control structures. Other activities of the program occupy a very small portion of the computational time and usually perform setup operations for the work on global data. This setup work is usually done between control structures. A microtasked version of the FFT subroutine follows:

```
      SUBROUTINE FFT2D(A,N,M)
      DIMENSION A(N,M),WORK(2048)
      CALL FFTINIT(N,WORK)
CMIC$ DO GLOBAL
      DO 100 I=1,M
        CALL FFTCOL(A(1,I),N,WORK)
  100 CONTINUE
      CALL FFTINIT(M,WORK)
CMIC$ DO GLOBAL
      DO 200 I=1,N
        CALL FFTROW(A(I,1),M,WORK)
  200 CONTINUE
      RETURN
      END
```

You have seen how two fairly simple subprograms can be microtasked. Real programs may be this simple, though in all likelihood they are not. Consequently, having a solid understanding of the concepts of microtasking is essential to keeping your coding fairly trouble free. First, you must understand how your data is stored.

Variables may be global or local; this is often referred to as the *scope* of the data. Global data items in Fortran are those variables that appear in COMMON blocks, SAVE statements, DATA statements, or in a subroutine's argument list. All other data items are local variables, and they are visible only within a particular subroutine. All local variables are stored on a separate stack for each processor that executes that subroutine. Data with global scope is known by the same address to each processor that enters the subroutine. The result of an operation performed on a global data item by one processor is known to all processors. Local data items exist in the stack for each processor that enters a microtasked subroutine. Thus, the result of an operation on local data is known only to the processor that performed it, because such results are not transmitted to other processors. Sometimes a subroutine has data defined as global data and it is not used in the results of the calling program, in which case it must be converted to local data.

Within a microtasked subroutine, the objective is to permit parallel processing of global data by imposing structure on the parts of the routine that modify it. This is done by allowing processors to enter the subroutine and proceed through it as they will. The fray itself starts and ends at a subroutine boundary so that the number of processors is indeterminate from the beginning of the subroutine until the RETURN statement. The fray can be entered by any number of processors that become available while it is being processed. The flow of control of these processors is restricted only by the control structures you define. To enforce order, define control structures within the body of the fray. You cannot control or know in advance how many processors will operate in any control structure.

A control structure bounds all the work that can proceed at one time, thereby defining the limits of parallelism. Because only one control structure can be active at a time, two items of work that depend on each other must reside in separate control structures. Within a control structure, the order of execution of the separate items of work (processes) is indeterminate. For example, a DO loop that modifies the rows of a two-dimensional array precedes a DO loop that modifies the columns and uses information already in the array to do so. The array is global data and must be modified inside a control structure. The second DO loop depends on the results of the first one, so the two loops must be in separate control structures. Within each DO loop, however, the order in which the rows or the columns are modified is immaterial.

All changes to global data must be made within control structures; otherwise, errors can result. If input or output must be done in a microtasked subroutine, it, too, must be done inside a control structure.

The area defined by a control structure contains processes that are independent of one another and can be executed in parallel. A process is the smallest unit of work that can be assigned to a processor. For example, if each iteration of a DO loop is independent of all other iterations, each iteration can be a process. Likewise, if two subroutines can be executed concurrently, each subroutine call can be a process and the two can be executed in parallel. Each process can run independently of all other processes within a control structure. Two processes that depend on each other in any way must reside in separate control structures.

4.2 ANALYZING A PROGRAM FOR MICROTASKING

For microtasking, you can use some of the same tools you use for vectorizing and add some new ones. The potential payoff of vectorizing is much greater than that of microtasking, which at most can speed up a program by a factor of nearly the number of processors.

Vectorize first, then microtask. In general, this means you should vectorize the innermost loops in a nested set of loops and microtask outer loops to make the best use of vectorization on multiple CPUs. Inner loops that are vectorizable can be both microtasked and vectorized using the long vector DO GLOBAL directives (LONG VECTOR, BY *expression*, and FOR *expression*), which effectively split the inner loop into one outer and one inner loop. Do not use a regular DO GLOBAL structure for an inner, vectorizable loop.

There are a number of tools to help you get ready to microtask. SPY and FLOWTRACE help determine what portions of the program are most worth working on, and FTREF helps determine the scope of data.

FLOWTRACE summarizes both the number of calls to subroutines and the portion of a program's time spent in those routines, and it provides a calling tree for the program. SPY samples while the program is executing and reports on the number of times it found the program working in certain label groupings. FLOWTRACE identifies subroutines in which the program spends its time; SPY, because it samples between statement labels, identifies frequently executed portions of those subroutines. (You cannot run FLOWTRACE on a multitasked program. Compile programs without PREMULT, comment out multitasking library calls, and change TSKSTARTS into calls to FLOWTRACE that examine the multitasked program in single-task mode.) FTREF provides a static calling tree and a cross-reference map. It shows in detail the use of both local variables and variables in COMMON blocks, helping you to see the scope of data.

FTREF, SPY, and FLOWTRACE are standard CRI products. For COS, they are described in the COS Performance Utilities Reference Manual. Flowtrace, **ftref**, and **prof** are implemented under UNICOS and are described in the UNICOS Performance Utilities Reference Manual.

4.3 MICROTASKING PREPROCESSOR DIRECTIVES

Inserting preprocessor directives is relatively simple, once you have done the data scoping. Other directives surround processes or provide locking mechanisms for synchronization. The appearance of a PROCESS, DO GLOBAL, or MICRO microtasking directive in a subroutine signals that a subroutine is to be microtasked.

4.3.1 CMIC$ GETCPUS *n*

This directive must appear in the main program before any microtasked work is done. It specifies the maximum number of processors permitted to work on a microtasked program, with *n* an integer constant or variable number of processors. The default value for *n* is the maximum number of physical CPUs available for your program.

4.3.2 CMIC$ RELCPUS

This directive specifies that the processors acquired for microtasking should be released back to the system. It is the reverse of the GETCPUS directive. This directive should be used when no microtasking is to be done for a long period of time or when the program is preparing to terminate.

This directive is optional; if it is not used, all processors acquired by the GETCPUS directive are held until the program terminates. When a STOP, END, or CALL EXIT statement is encountered, the microtasking slave processors are automatically released before the job step is terminated.

4.3.3 CMIC$ MICRO

This directive designates a subroutine to be microtasked and appears just before the SUBROUTINE statement. A subroutine introduced in this way becomes a microtasked subroutine, or fray. Executing a RETURN or END statement signals the end of multiprocessing work. Upon exit, only one processor returns to the calling routine. A function may not be microtasked, though it may, of course, be rewritten as a subroutine and then microtasked.

This directive is not required if the subroutine has either a PROCESS or a DO GLOBAL directive.†

Example:

```
CMIC$ MICRO
      SUBROUTINE TASKER
```

4.3.4 CMIC$ PROCESS

This directive marks the beginning of a control structure and signals that the code following it is a single process.

4.3.5 CMIC$ ALSO PROCESS

This directive marks the beginning of a process other than the first process inside a control structure and the end of the previous process. Fortran limitations restrict the number of ALSO PROCESS directives to about 100 per control structure. A PROCESS directive followed by any number of ALSO PROCESS directives implements a classic fork-and-join multitasking structure.

---

† Deferred COS implementation

4.3.6 CMIC$ END PROCESS

This directive marks the end of a process and the end of a control structure. PROCESS and END PROCESS directives can also be used to ensure single-processor execution of a portion of code. The single-threaded section goes in a control structure with just one process (that is, the section contains no ALSO PROCESS directive).

Example:

```
CMIC$ PROCESS
      DO 10 I= 1, 1000
        A(I) = X(I) * Y(I)
10    CONTINUE

CMIC$ ALSO PROCESS
      DO 20 I= 1, 1000
        B(I) = X(I) * Z(I)
20    CONTINUE

CMIC$ END PROCESS
```

4.3.7 CMIC$ DO GLOBAL

This directive marks the beginning of a control structure in which the iterations of a DO loop comprise all of the processes. The control structure is, therefore, a special case of the CMIC$ PROCESS...CMIC$ END PROCESS control structure (one in which each process is identical, except for the loop index). DO GLOBAL is probably the most commonly used control structure.

The statement following the CMIC$ DO GLOBAL directive is a DO statement. The end of the control structure is marked by the statement containing the label referred to in the DO statement; the DO GLOBAL control structure requires no preprocessor directive to close it.

DO GLOBAL directives may be used to create control structures within a DO loop, but the path through such control structures cannot be altered inside the microtasked subroutine. The DO loop being microtasked may not share its terminal statement at the end of the loop (such as CONTINUE) with any other loop. A DO GLOBAL statement may be nested within a DO loop, but only one DO GLOBAL can be executing at a time.

Loops using DO GLOBAL must have integers as initial, final, and step values.

Three variants of the DO GLOBAL directive are supplied to help you better balance microtasking and vectorization. These variants are described in the next three subsections.

Example:

```
CMIC$ DO GLOBAL
      DO 20 J= 1, 1000
       DO 10 I= 1, 1000
          A(I,J)= X(I,J) * Y(I,J)
10    CONTINUE
20    CONTINUE
```

4.3.8 CMIC$ DO GLOBAL LONG VECTOR†

This directive marks the beginning of a control structure that permits both vectorization and microtasking on an innermost DO loop. This structure divides a loop into processes of 64 iterations each, microtasking the groups and vectorizing the iterations. (One remainder group will have 64 or fewer iterations.)

This directive cannot provide a speedup if there are fewer than 64 iterations in the loop; the loop should be longer than 64 iterations, and it should be vectorizable. Two associated directives (DO GLOBAL BY and DO GLOBAL FOR) let you change the iteration group size, also known as the *chunking factor*. See subsection 3.5.2, Load Balancing, for a discussion of trade-offs associated with large and small chunking factors with regard to dynamic load balancing.

Example:

```
CMIC$ DO GLOBAL LONG VECTOR
      DO 100 K = 1, 4096
        A(K) = B(K) * C(K)
100   CONTINUE
```

This example divides the original loop into an inner and outer loop, each of 64 iterations.

4.3.9 CMIC$ DO GLOBAL BY *expression*†

This directive is the same as the DO GLOBAL LONG VECTOR directive except that the iterations are divided into groups of size *expression*.

---

† Deferred COS implementation

It divides a DO loop into an inner loop, with *expression* iterations, and an outer loop. The number of iterations in the outer loop is approximately the number of iterations in the original DO loop divided by *expression*. The inner loop is then vectorized and the outer loop microtasked. Setting *expression* to a multiple of 64 maximizes the vectorization performance.

You must ensure that the Fortran expression evaluates to an integer greater than zero. The expression is evaluated at run time and may change each time the DO loop is executed, but it cannot change between iterations of the same execution.

Example:

```
CMIC$ DO GLOBAL BY 1024
      DO 100 K = 1, 4096
        A(K) = B(K) * C(K)
100   CONTINUE
```

In this example, the 4096 iterations of the DO loop are divided into four pieces of 1024 each.

4.3.10 CMIC$ DO GLOBAL FOR *expression*†

This directive is the same as the DO GLOBAL LONG VECTOR directive, except that the iterations are divided into *expression* groups.

It divides a DO loop into an outer loop, with *expression* iterations, and an inner loop. The number of iterations in the inner loop is approximately the number of iterations in the original DO loop divided by *expression*. The inner loop is then vectorized and the outer loop microtasked.

Example:

```
CMIC$ DO GLOBAL FOR 4
      DO 100 K = 1, 4096
        A(K) = B(K) * C(K)
100   CONTINUE
```

This example specifies the number of iterations for the generated outer loop to be 4. The number of iterations for the inner loop is then 1024. The effect is same as the example for the DO GLOBAL BY directive in the previous subsection. The only difference is whether you want to specify the chunk size or the number of chunks.

† Deferred COS implementation

4.3.11 CMIC$ STOP ALL PROCESS

This directive provides a way to exit from both PROCESS and DO GLOBAL control structures without performing all of the processes or iterations. This directive forces all processors to complete work in a process if they are in one, then accept no more work, closing the control structure.

Processors resume work at the first statement after the end of the control structure. This will be either the statement after CMIC$ END PROCESS or the statement after the CONTINUE statement that ends the DO GLOBAL DO loop. For example, you may wish to end processing in a DO loop when a certain solution is found. If the solution is never found, the loop is executed some maximum number of iterations. STOP ALL PROCESS provides this graceful exit. Typically, the program will appear as in the following example.

Example:

```
CMIC$ DO GLOBAL
      DO 1 I = 1,10000
        . . .
        IF end-condition THEN
CMIC$ STOP ALL PROCESS
          GO TO 2
        ENDIF
        . . .
    1 CONTINUE
    2 CONTINUE
```

The previous section of code is portable. You must ensure that there is no work done between the statement that ends the DO loop and the statement at which processing resumes. You must also ensure that the statement number to which the single-processor version jumps and the one to which the microtasked version jumps (as a result of the STOP ALL PROCESS directive) are the same. The preprocessor does not catch errors you might make in using the STOP ALL PROCESS directive.

4.3.12 CMIC$ GUARD *n*

This directive marks the beginning of a critical region to be protected from concurrent execution. The GUARD directive is similar in purpose to the macrotasking LOCK routines. It occurs within a control structure or within a routine called from inside a control structure. The guarded portion of the program is not itself a control structure, however.

Only sections of code guarded by the same *n* are prevented from executing simultaneously. The *n* is an integer from 0 through 63, and, if *n* is greater than 63, its remainder when divided by 64 is used. Thus, GUARD 0 is equivalent to GUARD 64. GUARD with no *n* supplied constitutes an especially rapid guard, in which case no other guarded code can be executed simultaneously.

The GUARD must be used with care inside subroutines called by microtasked routines, because in such a situation, the preprocessor cannot determine whether one GUARD is nested inside another. You must ensure that this does not happen, because it leads to a deadlock. The preprocessor issues a warning message to alert you to this potential error when it encounters a GUARD directive inside a subroutine that is not microtasked.

4.3.13 CMIC$ END GUARD *n*

This directive marks the end of a critical region that is protected from concurrent execution.

```
CMIC$ DO GLOBAL
      DO 10 I= 1, N
        DO 20 I= 1, M
          LOCALSUM= LOCALSUM + A(I,J)
20    CONTINUE
CMIC$ GUARD
      GLOBLSUM= GLOBLSUM + LOCALSUM
CMIC$ END GUARD
10    CONTINUE
```

4.3.14 CMIC$ CONTINUE

This directive causes microtasking to continue within another subroutine. It appears within a microtasked subroutine but outside any control structure in the subroutine. It appears just before a call to another microtasked subroutine, and it directs that microtasking will continue in the called subroutine. The effect is as if the code in the called routine were brought into the calling routine.

This allows microtasking without substantially restructuring the original program. The CONTINUE directive cannot appear within a control structure, nor does it delineate a control structure.

In the following example, two DO loops are microtasked: one in subroutine SUB1 and one in subroutine SUB2.

Example:

```
      SUBROUTINE SUB1
CMIC$ DO GLOBAL
      DO 15 I = 1, 100
        DO 10 J = 1, 100
          DO 10 K = 1, 100
            (Many computations)
  10  CONTINUE
  15  CONTINUE
CMIC$ CONTINUE
      CALL SUB2
      RETURN
      END

CMIC$ MICRO
      SUBROUTINE SUB2
CMIC$ DO GLOBAL
      DO 25 I = 1, 100
        DO 20 J = 1, 100
          DO 20 K = 1, 100
            (Many computations)
  20  CONTINUE
  25  CONTINUE
      RETURN
      END
```

The effect is the same as if the DO 25 loop directive had appeared in place of the original call to SUB2. The CONTINUE directive may appear at more than one level. For example, SUB2 could have a call to SUB3, another microtasked routine, and that call could be preceded by a CONTINUE directive.

Subroutine calls from within microtasked routines need not necessarily be preceded by a CONTINUE directive. Within a control structure, calls to subroutines may not be preceded by a CONTINUE directive. Also, the subroutine called after a CONTINUE directive must be a microtasked routine; that is, it must contain a PROCESS, DO GLOBAL, or MICRO directive.

4.4 ACCESSING THE PREPROCESSOR

PREMULT is the microtasking preprocessor that interprets the preprocessing directives described in the previous subsection and rewrites your program. The PREMULT preprocessor is available under COS and UNICOS.

As with macrotasking, you must also include the MULTI statement under COS or the **multi** command under UNICOS to access the multitasking libraries.

4.4.1 INVOKING PREMULT UNDER COS

The following format describes the PREMULT control statement under COS:

```
PREMULT,I=idn,M=mdn,S=sdn,C=cdn,LONGNAM,CFT77,CALV2.
```

Parameter	Default	Keyword Alone	Description
idn	$IN	$IN	Input dataset to PREMULT
mdn	$MULTF	$MULTF	Output from PREMULT. This is the Fortran code that contains inserted microtasking primitives.
sdn	*mdn*	$MULTF	Output from PREMULT. Fortran with no microtasking primitives. Microtasked subroutines also exist in a nonmicrotasked version, so they may be called from a microtasked routine.
cdn	$MULTC	$MULTC	Output from PREMULT. CAL master routine for each microtasked subroutine.
LONGNAM	Not selected	Selected	By default, the preprocessor creates two subroutines from each one you microtask and appends to their names as much of MULT and SNGL as it can without making their names longer than 8 characters. Thus, JOE becomes JOEMULT and JOESNGL, and LONGJOE becomes LONGJOEM and LONGJOES. PREMULT aborts if it finds an 8-character subroutine name in a program. Selecting LONGNAM tells PREMULT to replace the last character of an 8-character subroutine name with S or M. You must ensure that subroutine names thus created are unique. PREMULT aborts if the original 8-character routine name ends in S or M.

Parameter	Default	Keyword Alone	Description
CFT77	Not selected	Selected	By default, the preprocessor generates code that is acceptable to the CFT compiler and that contains CDIR$ BLOCK directives. Selecting CFT77 tells PREMULT to generate code acceptable to the CFT77 compiler, containing CDIR$ SUPPRESS directives.
CALV2	Not selected	Selected	By default, the preprocessor generates code that is acceptable to the CAL Version 1 assembler and that uses a nonstandard method of defining and referencing a TASK COMMON block. Selecting CALV2 tells PREMULT to generate code that is acceptable to the CAL Version 2 assembler and that defines and references the TASK COMMON block by using the new SECTION pseudo-op.

The output datasets are rewound if their names are not $OUT. The input dataset is rewound if its name is not $IN. If an output dataset is $OUT, a space is inserted in column 1 of the output. The source is checked for improper nesting of directives. If errors are found in the source, the offending routines are sent to $OUT and the job is aborted. The option to separate the M and S output datasets lets you force the microtasked routines to a different compile unit; you can, for example, suppress the listing of the routines that contain the microtasking primitives.

Example:

```
MULTI.
PREMULT.
*.  PREMULT CREATES $MULTF (FORTRAN) AND $MULTC (CAL).
CFT,I=$MULTF,ALLOC=STACK.
CAL,I=$MULTC.
SEGLDR,GO,CMD='STACK=       ;HEAP=       '.
/EOF
. . .
Fortran program with CMIC$ microtasking directives.
. . .
```

Note the presence of the MULTI statement to access the multitasking libraries and that the compiler control statement, CFT in this case, requires the ALLOC=STACK parameter. In addition, the CAL assembly step requires the stack version of $SYSTXT, which may not be the default at your site. Check with the system administrator or a CRI site analyst.

4.4.2 INVOKING PREMULT UNDER UNICOS

The syntax for invoking PREMULT from a UNICOS command line is as follows. Blanks between the option (such as -s) and its argument are optional. Options can be specified in any order.

```
premult [-m mdn] [-s sdn] [-l] [-c cdn] [-F]  file.f
```

Parameter	Default	Description
-m *mdn*	multf.f	Output from PREMULT. This is the Fortran code that contains inserted microtasking primitives.
-s *sdn*	*mdn*	Output from PREMULT. This is the Fortran code with no microtasking primitives. Microtasked subroutines also exist in a nonmicrotasked version, so they may be called from a microtasked routine.
-l	Not selected	By default, the preprocessor creates two subroutines from each one you microtask and appends to their names as much of MULT and SNGL as it can without making their names longer than 8 characters. Thus, JOE becomes JOEMULT and JOESNGL, and LONGJOE becomes LONGJOEM and LONGJOES. PREMULT aborts if it finds an 8-character subroutine name in a program. Selecting -l tells PREMULT to replace the last character of an 8-character subroutine name with S or M. You must ensure that subroutine names thus created are unique. PREMULT aborts if the original 8-character routine name ends in S or M.
-c *cdn*	multc.s	Output from PREMULT. CAL master routine for each microtasked subroutine.
-F	Not selected	If present, this parameter indicates that CFT77 is the Fortran compiler. If this parameter is not selected, CFT is assumed.
file.f	None	Input file to PREMULT

The following example shows the sequence of UNICOS commands needed to run a microtasked CFT job named **prog.f**:

```
premult prog.f
cft -a stack multf.f
as multc.s
segldr -o prog multf.o multc.o segdir
```

The **premult** command creates two output files: **multf.f**, which contains Fortran code, and **multc.s**, which contains CAL code. The CAL assembly step (**as** command) requires the stack version of **asdef**, which may not be the default at your site. The file **segdir** contains six LIB directives that specify the directory containing the multitasking versions of the libraries. Check with the system administrator or a CRI site analyst for the locations of the multitasking libraries and stack **asdef** at your site.

4.4.3 NAMES RESERVED BY PREMULT

PREMULT must rewrite your Fortran program. To do so, it needs its own variables and a COMMON block. The following variable names are reserved for the use of PREMULT; you cannot use them in your program:

GDUMMY	GETCPUS	GROFF	GROFF1	GRON
GRON1	LPCHUNK	LPCON	LPEXIT	LPGDUMMY
LPST	LPST1	LPST64	LPSTOP	LPTSKCT
RELCPUS				

4.5 RULES TO FOLLOW

This subsection defines the rules for microtasking.

- Do not make any assumptions about the number of processors that will work on your program. A microtasked program must be designed to run correctly no matter how many processors are attached to it at run time. Test your microtasked program with one CPU and with more than one CPU. Compare results to ensure that they make sense and are consistent, and compare times across the runs to ensure that the number of processors is affecting execution time as anticipated.

- Do not modify shared data outside control structures. This is the most important rule of microtasking. Use whatever tools you need to identify shared data, then make sure that every statement that modifies a shared variable is in a control structure. You may find example programs that do change shared data outside control structures and seem to work. This is merely good fortune, however, and the next run may produce different answers. Do I/O only inside control structures. Performing I/O is like modifying shared data (the I/O files and their pointers). An example of a subroutine that gives unexpected results follows:

```
      SUBROUTINE TEST(parameters)
C        THIS SUBROUTINE TRIES TO KEEP TRACK OF WHICH ITERA-
C        TIONS OF A DO LOOP WERE ASSIGNED TO WHICH PROCESSOR
      DIMENSION IV(256)
      IS = 0
CMIC$ DO GLOBAL
      DO 100 I= 1,256
        IS = IS + 1
        IV(IS) = I
        . . .
        Real work of DO loop
        . . .
  100 CONTINUE
      PRINT *, (IV(I),I=1,IS)
      RETURN
      END
```

  In this subroutine, the system call for I/O pulls processors out in an unpredictable way. If the first processor that comes in is pulled out for an extended period by this system call, the other processors can repeatedly return to this statement and write a line. Extra, unexpected output can be generated, though it will not necessarily always happen, leading to inconsistent runs.

- A value given to a variable inside a control structure is not viable outside the control structure. In standard Fortran, the variable IS of the previous example always has the value 256 after DO loop 100. In microtasking, this value can vary from 0 to 256 for any particular processor. In fact, if the local value IS were summed for all processors, the total would be 256 after microtasking had issued DO loop 100 GLOBALLY.

- Do not nest control structures. A control structure defines the bounds of work that can be performed in parallel. Because the area of parallelism has already been defined by the outer control structure, an inner or nested control structure is already being performed in parallel and need not be defined as an area of parallel execution. Also, since the hardware shared registers are used to control the flow of processors through microtasked code and there is only one set of registers, any attempt to use these registers in a nested control structure would corrupt the values being maintained by the outer control structure. You may, however, use control structures inside a loop, an example of which follows:

```
      SUBROUTINE JOE
      DO 1 I = 1,N
CMIC$ DO GLOBAL
        DO 2 K = 1,M
        . . .
    2   CONTINUE
    1 CONTINUE
      RETURN
      END
```

The previous example is logically equivalent in microtasking to the following:

```
      SUBROUTINE JOE
      I = 1
CMIC$ DO GLOBAL
        DO 2 K = 1,M
        . . .
    2   CONTINUE
      I = 2
CMIC$ DO GLOBAL
        DO 3 K = 1,M
        . . .
    3   CONTINUE
      I = 3
      etc.
      RETURN
      END
```

The outer DO loop in the previous example cannot share a CONTINUE statement with the inner one. Set up separate CONTINUE statements for such outer DO loops.

- Each iteration of a microtasked DO loop (DO GLOBAL) must be genuinely independent, including the index variables. This requirement can occasionally force you to restructure the program. A matrix multiplication follows:

```
      SUBROUTINE MXM(A,B,C,L,M,N)
      DIMENSION A(L,M), B(M,N), C(L,N)
C
C        4-WAY UNROLLED MATRIX MULTIPLY ROUTINE FOR VECTOR
C        COMPUTERS.  M MUST BE A MULTIPLE OF FOUR.
C        CONTIGUOUS DATA ASSUMED.

C
C        J=1,4 PASS
C
      DO 100 K = 1,N
        DO 100 I = 1,L
          C(I,K) = (((A(I,1) * B(1,K)
     $             + A(I,2) * B(2,K)) + A(I,3) * B(3,K))
     $             + A(I,4) * B(4,K))
  100 CONTINUE
C
C        DO REMAINING J'S
C
      DO 110 J=5,M,4
        DO 110 K=1,N
          DO 110 I=1,L
            C(I,K) = C(I,K) + (((A(I,J) * B(J,K)
     $             + A(I,J+1) * B(J+1,K)) + A(I,J+2) * B(J+2,K))
     $             + A(I,J+3) * B(J+3,K))
  110 CONTINUE
C
      RETURN
      END
```

The inner loop of the triple loop vectorizes because the iterations are independent of I. The nesting order of the outer two loops does not matter for the single-threaded version. However, the outer loop has dependencies if it iterates on J, because each iteration modifies all elements of C. If, however, the nested loop is restructured so that the outer loop iterates on K, each processor that takes an iteration gets a distinct row of C to modify. Proper microtasking requires only the following minor modifications:

```
      SUBROUTINE MXM(A,B,C,L,M,N)
      DIMENSION A(L,M), B(M,N), C(L,N)
C
C        4-WAY UNROLLED MATRIX MULTIPLY ROUTINE FOR VECTOR
C        COMPUTERS. M MUST BE A MULTIPLE OF FOUR.
C        CONTIGUOUS DATA ASSUMED.
C
C        J=1,4 PASS
C
CMIC$ DO GLOBAL
      DO 100 K = 1,N
        DO 100 I = 1,L
          C(I,K) = (((A(I,1) * B(1,K)
     $             + A(I,2) * B(2,K)) + A(I,3) * B(3,K))
     $             + A(I,4) * B(4,K))
  100 CONTINUE
C
C        DO REMAINING J'S
C
CMIC$ DO GLOBAL
      DO 110 K=1,N
        DO 110 J=5,M,4
          DO 110 I=1,L
            C(I,K) = C(I,K) + (((A(I,J) * B(J,K)
     $             + A(I,J+1) * B(J+1,K)) + A(I,J+2) * B(J+2,K))
     $             + A(I,J+3) * B(J+3,K))
  110 CONTINUE
C
      RETURN
      END
```

Dependencies do not appear only in result data; local loop indices may also contain dependencies, as in the following incorrect example:

```
      SUBROUTINE EX(A,B,N,M)
      DIMENSION A(*), B(*)
C
      K = 0
CMIC$ DO GLOBAL
      DO 1  I = 1,N
        DO 2  J = 1,M
          A(J+K) = A(J+K) * B(J+K) * S
    2   CONTINUE
        K = K + M
    1 CONTINUE
      RETURN
      END
```

In the previous incorrect example, the value for K, when I=2, depends on K being calculated for I=1; therefore, K is a dependent variable as it is programmed here.

Correct the problem by changing K to be a function of I:

```
      SUBROUTINE EX(A,B,N,M)
      DIMENSION A(*), B(*)
C
CMIC$ DO GLOBAL
      DO 1 I = 1,N
        K = (I-1) * M
        DO 2 J = 1,M
          A(J+K) = A(J+K) * B(J+K) * S
    2   CONTINUE
    1 CONTINUE
      RETURN
      END
```

To avoid confusion and excessive index calculations, the program can also be rewritten and improved as follows:

```
      SUBROUTINE EX(A,B,N,M)
      DIMENSION A(M,N),B(M,N)
C
CMIC$ DO GLOBAL
      DO 1 I = 1,N
        DO 2 J = 1,M
          A(J,I) = A(J,I) * B(J,I) * S
    2   CONTINUE
    1 CONTINUE
        RETURN
        END
```

- The path through a microtasked subroutine's control structures cannot be determined by global variables that are modified within the subroutine. That is, all control structures in a microtasked subroutine must be accessible to all processors that enter the subroutine. Late processors must be able to check each control structure that has been entered by any processor to see if work in it has been completed. The following example violates this rule:

```
      SUBROUTINE ALLSUM(A,B,N)
      DIMENSION A(N,N),B(N,N)
      LOGICAL INIT
      DATA INIT /.FALSE./
C
      IF (.NOT. INIT) THEN
CMIC$ PROCESS
        DO 2 J = 1,N
          DO 1 I = 1,N
            A(I,J) = 0.0
    1     CONTINUE
    2   CONTINUE
        INIT = .TRUE.
CMIC$ END PROCESS
      END IF
C
CMIC$ DO GLOBAL
      DO 4 J = 1,N
        DO 3 J = 1,N
          A(I,J) = A(I,J) + B(I,J)
    3   CONTINUE
    4 CONTINUE
      RETURN
      END
```

On the subroutine call in which INIT is changed by the first processor entering, late processors can arrive after INIT is true. If they do, they do not have access to the PROCESS control structure to see whether the work in it has been completed. The following subroutine shows one way of reprogramming correctly:

```
      SUBROUTINE ALLSUM(A,B,N)
      DIMENSION A(N,N),B(N,N)
      LOGICAL INIT
      DATA INIT /.FALSE./
C
CMIC$ PROCESS
      IF (.NOT. INIT) THEN
        DO 2 J = 1,N
          DO 1 I = 1,N
            A(I,J) = 0.0
    1     CONTINUE
    2   CONTINUE
        INIT = .TRUE.
      END IF
CMIC$ END PROCESS
C
```

```
CMIC$ DO GLOBAL
      DO 4 J = 1,N
        DO 3 J = 1,N
          A(I,J) = A(I,J) + B(I,J)
    3   CONTINUE
    4 CONTINUE
      RETURN
      END
```

Moving the PROCESS and END PROCESS out to bracket the IF statement causes single threading through the section of code that does the initializing. The first processor to arrive gets the initializing work, and all other processors arrive at the control structure to check for work but are prevented from entering because of the single threading. On subsequent calls to the subroutine, INIT is true. In that case, the first processor in gets into the single-threaded section as before, but it drops through immediately because the IF test fails. The second and later processors do not get in, as before.

The following example is an even better way to do the same work; the value of INIT is set and reset in the calling routine:

```
      SUBROUTINE ALLSUM(A,B,N,INIT)
      DIMENSION A(N,N),B(N,N)
      LOGICAL INIT
C
      IF (.NOT. INIT) THEN
CMIC$ DO GLOBAL
        DO 2 J = 1,N
          DO 1 I = 1,N
            A(I,J) = 0.0
    1     CONTINUE
    2   CONTINUE
      END IF
C
CMIC$ DO GLOBAL
      DO 4 J = 1,N
        DO 3 J = 1,N
          A(I,J) = A(I,J) + B(I,J)
    3   CONTINUE
    4 CONTINUE
      RETURN
      END
```

- Use GUARD only inside control structures. GUARD prevents multiple processors from updating shared data, such as a global counter, simultaneously. Because GUARD directives protect the updating of shared data and because shared data should only be modified in control structures, the GUARD directive is useful only within control structures.

GUARD may be used in subroutines called from within a control structure. You are responsible for using it correctly in this case, because the preprocessor cannot check your usage. PREMULT generates a warning message to alert you of your responsibility.

- Verify that your program works and that it is being microtasked. You can test a microtasked program for correctness on just one CPU, but doing that does not demonstrate all potential problems that converting to microtasking can introduce. For thorough testing, run the original program on one CPU, then run the microtasked program on one CPU, then run the microtasked version on multiple CPUs, both batch and dedicated. Look at the wall-clock and CPU times for all three versions to see whether they are appropriate, given the percentage of the job that has been microtasked. Accounting of CPU use during microtasking is poor, if not invalid, for systems before COS release 1.16. Consequently, on systems predating COS release 1.16, use PERFMON (as explained in the next subsection) to obtain accurate CPU timing. Use a dedicated system to obtain optimal wall-clock timings.

These rules have been derived from observing mistakes that people make when they first try microtasking. If you follow these rules, you should be able to microtask successfully. For reference, a brief restatement of the rules follows:

- Make no assumptions about how many processors you will get.
- Modify shared data only inside control structures.
- Local variables set inside control structures are not viable outside them.
- Do not nest control structures.
- DO GLOBAL structures inside a larger DO loop must not share a CONTINUE statement with the larger DO loop.
- Iterations of microtasked DO loops must be genuinely independent.
- Global variables modified within control structures cannot determine the path through the subroutine.
- Use GUARD only inside control structures.
- Verify a microtasked program for correctness and performance.

4.6 PERFORMANCE OF MICROTASKED PROGRAMS

The aim of microtasking is to improve the wall-clock execution time of a program by close to N, where N is the number of processors called in CMIC$ GETCPUS. The low overhead of microtasking makes this more attainable than it is with macrotasking. The total CPU time for all processors of the parallel code should not be much higher than that of the single-processor version. In fact, the microtasked program, running on one CPU, should never be more than 5% slower than the single-processor version of the program.

There are a number of ways to measure program performance. On a dedicated system, straightforward comparison of original and microtasked wall-clock times will show the overall speedup due to microtasking. The job's log file gives statistics for each CPU, and the hardware performance monitor of the CRAY X-MP computer system can provide a more complete breakdown. The PERFMON utility, which is available only under COS, lets you collect information about your program from the hardware performance monitor. It can show you how microtasking affects things other than wall-clock and CPU time, such as memory contention.

Until COS version 1.16, PERFMON provides the only way to get accurate statistics on CPU use. PERFMON,ON=1. gives the time waiting on a semaphore for each processor. This is the time that the processor is not actually being used. Subtracting this waiting-on-a-semaphore time from the CPU time on all processors should make the CPU time consistent from run to run. This provides a workaround for the accounting problem mentioned for systems running versions of COS prior to 1.16. PERFMON is probably the best tool to use for this because it gives a lot of information without requiring additional programming.

The multitasking history trace buffers contain information that you may analyze to determine the performance of your job.

4.7 LONGER EXAMPLES

This subsection contains examples longer than the ones used in the text for illustration. In example 1, the subroutine NONBON is a time-consuming subprogram from a large molecular modeling system. It illustrates a number of the rules stated previously. The original subroutine is given first.

Example 1:

```
      SUBROUTINE NONBON(NATOM,NPAIR,IAR1,IAR2,IAC,ICO,X,F,CN1,
     +                  CN2,ASOL,BSOL,HBCUT,CG,XCHRG,ENB,EHB,EEL,
     +                  DIELD,NTYPES,NDRV)
C
C     GEORGE SEIBEL
C     DEPT OF PHARM CHEM
C     UCSF
C     SAN FRANCISCO, CA  94143
C     ----------------------------------------------------------------
C
      LOGICAL DIELD
C
      DIMENSION IAR1(*),IAR2(*),IAC(*),ICO(NTYPES,*),CG(*)
      DIMENSION XCHRG(*),CN1(*),CN2(*),ASOL(*),BSOL(*),HBCUT(*)
C
      DIMENSION X(3,*),F(3,*)
C
C     -- SCRATCH ARRAYS --
C
      DIMENSION IC(2500),XIJ(2500),YIJ(2500),ZIJ(2500),
     +          R2(2500),R6(2500),R10(2500),R12(2500),
     +          G(2500),DF(2500),DF1(2500),DF2(2500),HF1(2500),
     +          HF2(2500),VF1(2500),VF2(2500),FX(2500),FY(2500),
     +          FZ(2500),FXTEMP(2500),FYTEMP(2500),FZTEMP(2500)
C
C     -- POINTER ARRAYS --
C
      INTEGER P(2500)
C
      ENB = 0.0E+00
      EEL = 0.0E+00
      EHB = 0.0E+00
      LPACK = 1
C
C     LOOP OVER ALL ATOMS I IN THE SYSTEM
C
      DO 1000 I = 1, NATOM - 1
C
      NVDW = IAR1(I)
      NHB = IAR1(I+NATOM)
      NPR = NVDW + NHB
      NPACK = 1              ! MUST BE SET IN CASE WE JUMP OUT OF LOOP
C
      IF (NPR.EQ.0) GO TO 800
C
C     LOAD ATOM POINTER ARRAY P() FROM PACKED PAIRLIST
C
```

```
      NPACK = NPR / 4 + 1
      CALL UNPACK(IAR2(LPACK),16,P,4*NPACK)
C
      IF (DIELD) THEN
C
C     -- DISTANCE DEPENDENT DIELECTRIC --
C
         DO 300 J = 1, NPR
            IC(J) = ICO(IAC(I),IAC(P(J)))
            XIJ(J) = X(1,I) - X(1,P(J))
            YIJ(J) = X(2,I) - X(2,P(J))
            ZIJ(J) = X(3,I) - X(3,P(J))
            R2(J) = 1.0E0 / (XIJ(J)**2 + YIJ(J)**2 + ZIJ(J)**2)
            G(J) = CG(I) * CG(P(J)) * R2(J)
            DF2(J) = -(G(J) + G(J))
 300     CONTINUE
C
      ELSE
C
C     -- CONSTANT DIELECTRIC --
C
         DO 250 J = 1, NPR
            IC(J) = ICO(IAC(I),IAC(P(J)))
            XIJ(J) = X(1,I) - X(1,P(J))
            YIJ(J) = X(2,I) - X(2,P(J))
            ZIJ(J) = X(3,I) - X(3,P(J))
            R2(J) = 1.0E0 / (XIJ(J)**2 + YIJ(J)**2 + ZIJ(J)**2)
            G(J) = CG(I) * CG(P(J)) * SQRT(R2(J))
            DF2(J) = -G(J)
 250     CONTINUE
C
      ENDIF
C
      EEL = EEL + SSUM(NPR,G,1)
C
C     -- VDW 6 - 12 POTENTIAL --
C
      IF (NVDW .GT. 0) THEN
         DO 350 J = 1, NVDW
            R6(J) = R2(J)**3
            R12(J) = R6(J) * R6(J)
            VF1(J) = CN1(IC(J)) * R12(J)
            VF2(J) = CN2(IC(J)) * R6(J)
            DF1(J) = -12.0E0 * VF1(J) + 6.0E0 * VF2(J)
            DF(J) = (DF1(J) + DF2(J)) * R2(J)
 350     CONTINUE
C
         ENB = ENB + SSUM(NVDW,VF1,1) - SSUM(NVDW,VF2,1)
C
      ENDIF
```

```
C
C      -- HBOND PAIRS 10 - 12 POTENTIAL --
C
       IF (NHB .GT. 0) THEN
C
          DO 450 J = NVDW + 1, NPR
             IC(J) = IABS(IC(J))
             R10(J) = R2(J)**5


             HF1(J) = ASOL(IC(J)) * R10(J) * R2(J)
             HF2(J) = BSOL(IC(J)) * R10(J)
             DF1(J) = -12.0E0 * HF1(J) + 10.0E0 * HF2(J)
             DF(J) = (DF1(J) + DF2(J)) * R2(J)
 450      CONTINUE
C
          EHB = EHB + SSUM(NHB,HF1(NVDW+1),1) - SSUM(NHB,HF2(NVDW+1),1)
C
       ENDIF
C
C      -- UPDATE THE FORCE ARRAY --
C
       IF (NDRV.LE.0) GO TO 1000

       DO 500 J = 1, NPR
          FX(J) = XIJ(J) * DF(J)
          FY(J) = YIJ(J) * DF(J)
          FZ(J) = ZIJ(J) * DF(J)
C       TEMPORARIES USED HERE TO ACCOMPLISH VECTORIZING
          FXTEMP(J) = F(1,P(J)) + FX(J)
          FYTEMP(J) = F(2,P(J)) + FY(J)
          FZTEMP(J) = F(3,P(J)) + FZ(J)
 500   CONTINUE

       DO 700 J = 1, NPR
          F(1,P(J)) = FXTEMP(J)
          F(2,P(J)) = FYTEMP(J)
          F(3,P(J)) = FZTEMP(J)
 700   CONTINUE

       DUMX = SSUM(NPR,FX,1)
       DUMY = SSUM(NPR,FY,1)
       DUMZ = SSUM(NPR,FZ,1)

       F(1,I) = F(1,I) - DUMX
       F(2,I) = F(2,I) - DUMY
       F(3,I) = F(3,I) - DUMZ

  800 CONTINUE
       LPACK = LPACK + NPACK
 1000 CONTINUE
       RETURN
       END
```

CMIC$ GETCPUS must be added in the main program, and the subroutine is rewritten as follows. The changes to the program are highlighted with boldface type.

```
CMIC$ MICRO
      SUBROUTINE NONBON(NATOM,NPAIR,IAR1,IAR2,IAC,ICO,X,F,CN1,
     +                  CN2,ASOL,BSOL,HBCUT,CG,XCHRG,ENB,EHB,EEL,
     +                  DIELD,NTYPES,NDRV)
C
C     GEORGE SEIBEL
C     DEPT OF PHARM CHEM
C     UCSF
C     SAN FRANCISCO, CA  94143
C     ------------------------------------------------------------------
C
      LOGICAL DIELD
C
C     -- ADD PARAMETER FOR DIMENSIONING LPACK
C
      PARAMETER (MAXATOMS=32000)
C
C
      DIMENSION IAR1(*),IAR2(*),IAC(*),ICO(NTYPES,*),CG(*)
      DIMENSION XCHRG(*),CN1(*),CN2(*),ASOL(*),BSOL(*),HBCUT(*)
C
      DIMENSION X(3,*),F(3,*)
C
C     -- SCRATCH ARRAYS --
C
      DIMENSION IC(2500),XIJ(2500),YIJ(2500),ZIJ(2500),
     +          R2(2500),R6(2500),R10(2500),R12(2500),
     +          G(2500),DF(2500),DF1(2500),DF2(2500),HF1(2500),
     +          HF2(2500),VF1(2500),VF2(2500),FX(2500),FY(2500),
     +          FZ(2500),FXTEMP(2500),FYTEMP(2500),FZTEMP(2500)
C
C     -- SCRATCH ARRAY FOR MICROTASKING --
C
      DIMENSION LPACK(MAXATOMS)
C
C     -- POINTER ARRAYS --
C
      INTEGER P(2500)
C
C     -- ONLY ONE PROCESS TO EXECUTE INITIALIZATION --
C     -- OF THESE GLOBAL VARIABLES                  --
C
CMIC$ PROCESS
      ENB = 0.0E+00
      EEL = 0.0E+00
      EHB = 0.0E+00
CMIC$ END PROCESS
```

```
C
C      -- INITIALIZE LOOP CONTROL FOR MICROTASKING     --
C      -- LPACK IS CHANGED TO AN ARRAY AND CALCULATED --
C      -- IN ADVANCE FOR ALL I BECAUSE OF LOOP        --
C      -- DEPENDENCIES.                               --
C
       LPACK(1) = 1
       DO 50 I = 1, NATOM-1
       LPACK(I+1) = LPACK(I) + ((IAR1(I) + IAR1(I + NATOM))/4 + 1)
    50 CONTINUE
C
C      LOOP OVER ALL ATOMS I IN THE SYSTEM
C
CMIC$ DO GLOBAL
       DO 1000 I = 1, NATOM - 1
C
C      INITIALIZE LOCAL VARIABLES FOR MICROTASKING
C
       EELLOC = 0.0
       ENBLOC = 0.0
       EHBLOC = 0.0
C
       NVDW = IAR1(I)
       NHB = IAR1(I+NATOM)
       NPR = NVDW + NHB
C
C      -- THIS IS NOW TAKEN CARE OF ABOVE
C       NPACK = 1              ! MUST BE SET IN CASE WE JUMP OUT OF LOOP
C
       IF (NPR.EQ.0) GO TO 800
C
C      LOAD ATOM POINTER ARRAY P() FROM PACKED PAIRLIST
C
       NPACK = NPR / 4 + 1
       CALL UNPACK(IAR2(LPACK(I)),16,P,4*NPACK)
C
       IF (DIELD) THEN
C
C      -- DISTANCE DEPENDENT DIELECTRIC --
C
          DO 300 J = 1, NPR
             IC(J) = ICO(IAC(I),IAC(P(J)))
             XIJ(J) = X(1,I) - X(1,P(J))
             YIJ(J) = X(2,I) - X(2,P(J))
             ZIJ(J) = X(3,I) - X(3,P(J))
             R2(J) = 1.0E0 / (XIJ(J)**2 + YIJ(J)**2 + ZIJ(J)**2)
             G(J) = CG(I) * CG(P(J)) * R2(J)
             DF2(J) = -(G(J) + G(J))
 300      CONTINUE
C
```

```
      ELSE
C
C     -- CONSTANT DIELECTRIC --
C
         DO 250 J = 1, NPR
            IC(J) = ICO(IAC(I),IAC(P(J)))
            XIJ(J) = X(1,I) - X(1,P(J))
            YIJ(J) = X(2,I) - X(2,P(J))
            ZIJ(J) = X(3,I) - X(3,P(J))
            R2(J) = 1.0E0 / (XIJ(J)**2 + YIJ(J)**2 + ZIJ(J)**2)
            G(J) = CG(I) * CG(P(J)) * SQRT(R2(J))
            DF2(J) = -G(J)
 250     CONTINUE
C
      ENDIF
C
C     EEL = EEL + SSUM(NPR,G,1)
C     -- MAKE LOCAL EEL FOR EACH PROCESSOR; ACCUMULATE LATER --
C
      EELLOC = SSUM(NPR,G,1)
C
C     -- VDW 6 - 12 POTENTIAL --
C
      IF (NVDW .GT. 0) THEN
         DO 350 J = 1, NVDW
            R6(J) = R2(J)**3
            R12(J) = R6(J) * R6(J)
            VF1(J) = CN1(IC(J)) * R12(J)
            VF2(J) = CN2(IC(J)) * R6(J)
            DF1(J) = -12.0E0 * VF1(J) + 6.0E0 * VF2(J)
            DF(J) = (DF1(J) + DF2(J)) * R2(J)
 350     CONTINUE
C
C         ENB = ENB + SSUM(NVDW,VF1,1) - SSUM(NVDW,VF2,1)
C     -- MAKE LOCAL ENB FOR EACH PROCESSOR; ACCUMULATE LATER --
C
         ENBLOC = SSUM(NVDW,VF1,1) - SSUM(NVDW,VF2,1)
C
      ENDIF
C
C     -- HBOND PAIRS 10 - 12 POTENTIAL --
C
      IF (NHB .GT. 0) THEN
C
```

```
         DO 450 J = NVDW + 1, NPR
            IC(J) = IABS(IC(J))
            R10(J) = R2(J)**5
            HF1(J) = ASOL(IC(J)) * R10(J) * R2(J)
            HF2(J) = BSOL(IC(J)) * R10(J)
            DF1(J) = -12.0E0 * HF1(J) + 10.0E0 * HF2(J)
            DF(J) = (DF1(J) + DF2(J)) * R2(J)
 450     CONTINUE
C
C         EHB = EHB + SSUM(NHB,HF1(NVDW+1),1) - SSUM(NHB,HF2(NVDW+1),1)
C     -- MAKE LOCAL EHB FOR EACH PROCESSOR; ACCUMULATE LATER --
C
         EHBLOC = SSUM(NHB,HF1(NVDW+1),1) - SSUM(NHB,HF2(NVDW+1),1)
C
      ENDIF
C
C
C
C     -- UPDATE THE FORCE ARRAY --
C
      IF (NDRV.LE.0) THEN
C
C     UPDATE EEL, ENB, EHB BEFORE EXITING.  GUARD IS REQUIRED
C     TO PREVENT OTHER PROCESSORS FROM UPDATING AT SAME TIME
C
CMIC$ GUARD
        EEL = EEL + EELLOC
        ENB = ENB + ENBLOC
        EHB = EHB + EHBLOC
CMIC$ END GUARD
        GO TO 1000
C
      ELSE
C
C     IF FORCE ARRAY MUST BE UPDATED, PLACE CODE TO BE GUARDED
C     TOGETHER TO MINIMIZE TURNING GUARDS ON AND OFF.

        DO 500 J = 1, NPR
          FX(J) = XIJ(J) * DF(J)
          FY(J) = YIJ(J) * DF(J)
          FZ(J) = ZIJ(J) * DF(J)
 500    CONTINUE
C     THIS SECTION MOVED UP FROM BELOW
        DUMX = SSUM(NPR,FX,1)
        DUMY = SSUM(NPR,FY,1)
        DUMZ = SSUM(NPR,FZ,1)
C
CMIC$ GUARD
C
        EEL = EEL + EELLOC
        ENB = ENB + ENBLOC
        EHB = EHB + EHBLOC
```

```
C
        DO 600 J = 1, NPR
C     TEMPORARIES USED HERE TO ACCOMPLISH VECTORIZING
          FXTEMP(J) = F(1,P(J)) + FX(J)
          FYTEMP(J) = F(2,P(J)) + FY(J)
          FZTEMP(J) = F(3,P(J)) + FZ(J)
 600    CONTINUE

        DO 700 J = 1, NPR
          F(1,P(J)) = FXTEMP(J)
          F(2,P(J)) = FYTEMP(J)
          F(3,P(J)) = FZTEMP(J)
 700    CONTINUE

        F(1,I) = F(1,I) - DUMX
        F(2,I) = F(2,I) - DUMY
        F(3,I) = F(3,I) - DUMZ
CMIC$ END GUARD
C
      ENDIF
  800 CONTINUE

C     -- NO LONGER NEEDED
C      LPACK = LPACK + NPACK
 1000 CONTINUE
      RETURN
      END
```

Example 2 is part of a fluid dynamics program in which five subroutines are microtasked. The example contains the version of the program before microtasking on the left and the version after microtasking on the right.

Example 2:

Before Microtasking

```
C     MAIN PROGRAM

      DO 10 N=1, NEND
       CALL EULER
10    CONTINUE
      STOP
      END
```

After Microtasking

```
C     MAIN PROGRAM
CMIC$ GETCPUS
      DO 10 N=1, NEND
       CALL EULER
10    CONTINUE
      STOP
      END
```

Before Microtasking

```
      SUBROUTINE EULER

      DO 10 K=1, KL
       DO 10 J=1, JL
        DO 10 I=1, IL
C        Many computations
10    CONTINUE

      CALL FILTER

      CALL TSL

      CALL BC

      RETURN
      END



      SUBROUTINE FILTER

      DO 10 K=1, KL
       DO 10 J=1, JL
        DO 10 I=1, IL
C       Many computations
10    CONTINUE
      RETURN
```

After Microtasking

```
CMIC$ MICRO
      SUBROUTINE EULER
CMIC$ DO GLOBAL
      DO 10 K=1, KL
       DO 10 J=1, JL
        DO 10 I=1, IL
C        Many computations
10    CONTINUE
CMIC$ CONTINUE
      CALL FILTER
CMIC$ CONTINUE
      CALL TSL
CMIC$ PROCESS
      CALL BC
CMIC$ END PROCESS
      RETURN
      END



CMIC$ MICRO
      SUBROUTINE FILTER
CMIC$ DO GLOBAL
      DO 10 K=1, KL
       DO 10 J=1, JL
        DO 10 I=1, IL
C       Many computations
10    CONTINUE
      RETURN
```

Before Microtasking

```
      SUBROUTINE TSL

      CALL TSL1

      CALL EMUTURB

      DO 10 K=1, KL
       DO 10 J=1, JL
        DO 10 I=1, IL
C     Many computations
10    CONTINUE
      RETURN
      END
```

After Microtasking

```
CMIC$ MICRO
      SUBROUTINE TSL
CMIC$ CONTINUE
      CALL TSL1
CMIC$ CONTINUE
      CALL EMUTURB
CMIC$ DO GLOBAL
      DO 10 K=1, KL
       DO 10 J=1, JL
        DO 10 I=1, IL
C     Many computations
10    CONTINUE
      RETURN
      END
```

Before Microtasking

```
      SUBROUTINE TSL1

      DO 10 K=1, KL
       DO 10 J=1, JL
        DO 10 I=1, IL
C     Many computations
10    CONTINUE
      RETURN
      END



      SUBROUTINE EMUTURB

      DO 10 K=1, KL
       DO 10 J=1, JL
        DO 10 I=1, IL
C     Many computations
10    CONTINUE
      RETURN
      END
```

After Microtasking

```
CMIC$ MICRO
      SUBROUTINE TSL1
CMIC$ DO GLOBAL
      DO 10 K=1, KL
       DO 10 J=1, JL
        DO 10 I=1, IL
C     Many computations
10    CONTINUE
      RETURN
      END


CMIC$ MICRO
      SUBROUTINE EMUTURB
CMIC$ DO GLOBAL
      DO 10 K=1, KL
       DO 10 J=1, JL
        DO 10 I=1, IL
C     Many computations
10    CONTINUE
      RETURN
      END
```

5. MACROTASKING

Macrotasking applies multiple processors to a Fortran job at the subroutine level. Whereas microtasking is often appropriate with jobs that require relatively little CPU time or that have small granularity, macrotasking was designed for long-running jobs of larger granularity, normally in a dedicated environment. Macrotasking is less useful on small jobs because it has a higher overhead than microtasking.

With macrotasking, you explicitly partition your job into tasks, each of which is eligible to run on a CPU. Typically, these tasks may take the form of different subroutines that are able to execute concurrently, or they may involve separate invocations of the same subroutine.

Figure 5-1 represents a program that has two sets of subroutines that can execute at the same time.

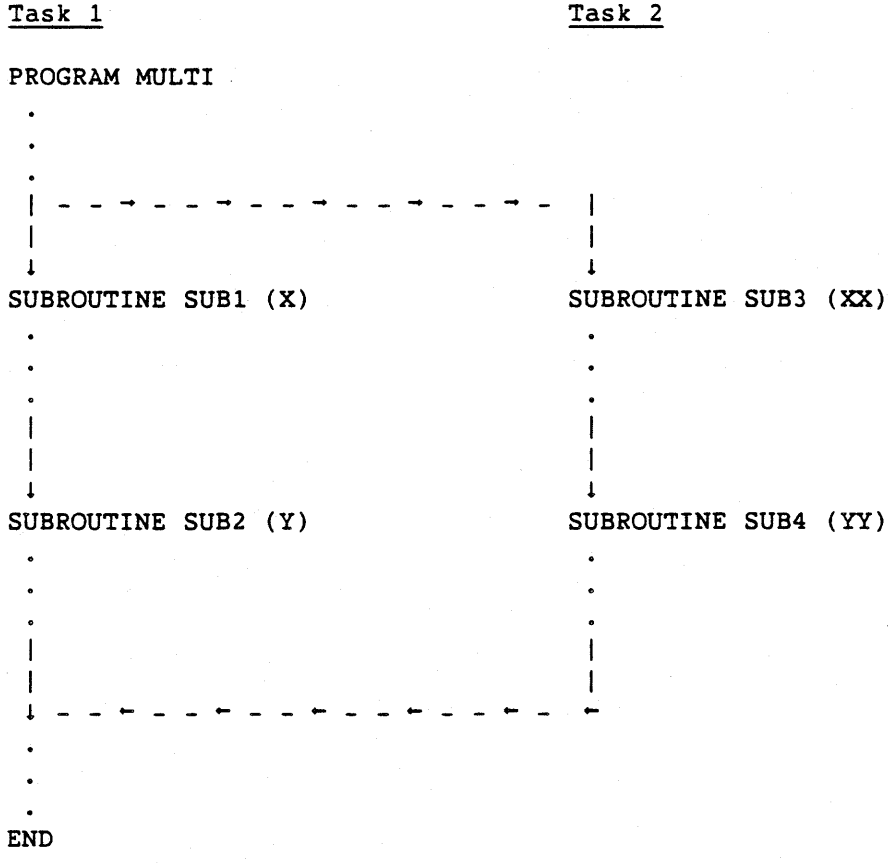


Figure 5-1. Macrotasking with Dissimilar Subroutines

This example is easily multitasked if the subroutines in task 2 use different data than those in task 1 or if the sequence in which that data is accessed is not important. If the order in which they access common data is important, however, safeguards are required. See subsection 5.2.4, Locks, where such safeguards are described.

Figure 5-2 shows subroutines running as separate tasks that both call a third subroutine, named COMSUB here.

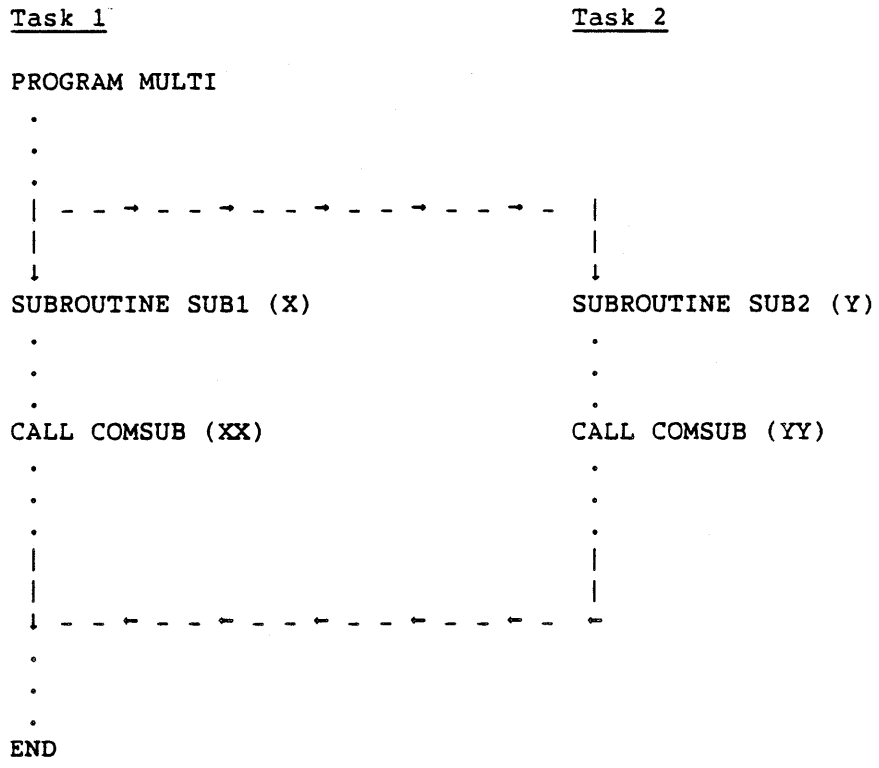


Figure 5-2. Macrotasking with a Common Subroutine

There is an excellent chance that bad data could result in this example if the two executions of COMSUB overlap. Methods of protecting the data are available, however.

5.1 PARALLELISM AND TASKS

The basic macrotasking routines deal with tasks. This subsection deals with tasks and their relationships. Subsequent subsections discuss the macrotasking routines that handle locks, events, and barriers.

5.1.1 TASKS

A task consists of code and data that can be scheduled for execution on a CPU. A Fortran program can have any number of tasks, all of which are assigned the same priority and memory characteristics as the job itself.

A task is defined as starting execution at a Fortran entry point (typically a subroutine), and it can call other subroutines during its execution. A task completes when it executes a RETURN statement in the subroutine in which it began execution, when it executes a STOP statement or equivalent operation, or when its execution is aborted because of an error condition. When it ends by a RETURN, STOP, END, or CALL EXIT, only that task ends. When it terminates for an error condition or CALL ABORT, all other tasks are stopped as soon as possible.

Any program executed under COS or UNICOS has an initial root task created by the system. This task suffices for nonmultitasked programs and products, and these codes do not require modification to run on a system supporting multitasking. Multitasking within a program begins as soon as the program explicitly creates another task through a call to TSKSTART.

5.1.2 TASK STATES

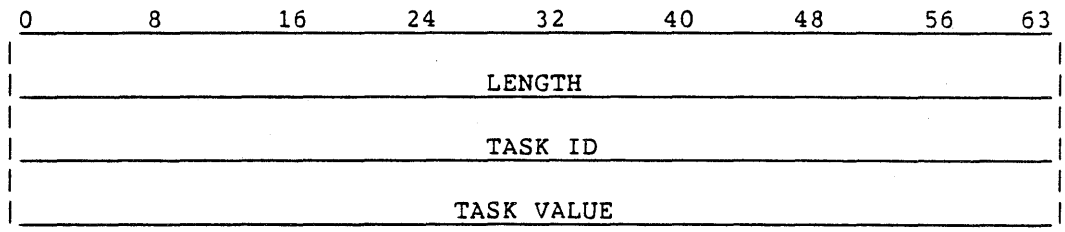
The library routines described in this section view tasks as being in one of two states: existing or not existing. A task exists from the time it is created until the time it completes execution. Among nonexistent tasks, no distinction is made between a task that has never existed and a task that has completed execution.

5.1.3 TASK RELATIONSHIPS

No default relationships exist between tasks. You make and enforce decisions to use specific intertask relationships (such as co-routines or a parent-child relationship).

5.1.4 TASK CONTROL ARRAY

An integer-defined task control array, which must be built by the user program, represents each user-created task. The array can be either 2 or 3 words in length. The array structure is as follows:



Field	Word	Bits	Description
LENGTH	1	0-63	Length of the array in words. Set the length to a value of 2 or 3, depending on the optional use of the TASK VALUE field. Set the LENGTH field before creating the task.
TASK ID	2	0-63	Task identifier assigned by the multitasking library when a task is created. This identifier is unique among active tasks within the job step. The multitasking library uses this field for task identification, but it is of limited use to user programs and should never be changed.
TASK VALUE	3	0-63	Optional value that you can set to any value before creating the task. If you use TASK VALUE, set LENGTH to 3. The task value can be used for any purpose. Suggested values include a task name or identifier that you generate or a pointer to a task local storage area. During execution, a task can retrieve this value with the TSKVALUE subroutine.

Example:

```
      PROGRAM MULTI
      INTEGER TASKARY (3)
C
C        SET TASKARY PARAMETERS
      TASKARY(1)=3
      TASKARY(3)='TASK 1'H
      ...
      END
```

5.1.5 TSKSTART

TSKSTART initiates a task. You invoke TSKSTART by using the following format:

```
CALL TSKSTART (taskarray,name[,list])
```

taskarray Task control array used for this task. Word 1 must be set and word 3, if used, must also be set. On return, word 2 is set to a unique task identifier that the program must not change.

name External entry point at which task execution begins. This name must be declared EXTERNAL in the program or subroutine making the call to TSKSTART. CFT77 and CFT do not allow a subroutine to use its own name in this parameter.

list Optional list of arguments passed to the new task. This list can be of any length. In general, do not use this list of arguments (see caution following).

****

CAUTION

Arguments passed in *list* are passed by address to the newly started task. As a result, the arguments become shared data whose subsequent use by different tasks must be synchronized.

Do not pass expressions as arguments in *list*. The compiler stores the computed expression on the stack at run time and can reuse the storage any time after TSKSTART returns to the calling task (even though the started task may not have executed).

Similarly, you should not pass local variables as arguments in *list* if the subroutine that calls TSKSTART does not also call TSKWAIT for the same task. Local variables are stored on the stack. A subroutine's stack space is reused after a return, so the arguments' storage space may have been reused by the time the arguments are accessed by the new task.

****

CFT analyzes the arguments to TSKSTART and issues warning messages for invalid or potentially dangerous arguments such as the following:

- An argument in *list* is assigned to a register.
- An argument in *list* is a local stack variable.
- An argument in *list* is an expression.
- *name* is not declared to be external.
- *taskarray* is not an integer array.

A call to TSKSTART is identical to CALL *name* [(*list*)], except that *name* is executed as a task instead of a subroutine.

Example:

```
      PROGRAM MULTI
      INTEGER TASK1ARY(3),TASK2ARY(3)
      EXTERNAL PLLEL
      REAL DATA(40000)
C
C        LOAD DATA ARRAY FROM SOME OUTSIDE SOURCE
      ...
C        CREATE TASK TO EXECUTE FIRST HALF OF THE DATA
      TASK1ARY(1)=3
      TASK1ARY(3)='TASK 1'H
C
      CALL TSKSTART (TASK1ARY,PLLEL,DATA(1),20000)
C
C        CREATE TASK TO EXECUTE SECOND HALF OF THE DATA
      TASK2ARY(1)=3
      TASK2ARY(3)='TASK 2'H
C
      CALL TSKSTART (TASK2ARY,PLLEL,DATA(20001),20000)
      ...
      END
```

5.1.6 TSKWAIT

TSKWAIT waits for the indicated task to complete execution.

Format:

```
CALL TSKWAIT (taskarray)
```

taskarray Task control array

Example:

```
      PROGRAM MULTI
      INTEGER TASK1ARY(3),TASK2ARY(3)
      EXTERNAL PLLEL
      REAL DATA(40000)
C
C        LOAD DATA ARRAY FROM SOME OUTSIDE SOURCE
      ...
C        CREATE TASK TO EXECUTE FIRST HALF OF THE DATA
      TASK1ARY(1)=3
      TASK1ARY(3)='TASK 1'H
C
      CALL TSKSTART (TASK1ARY,PLLEL,DATA(1),20000)
C
C        CREATE TASK TO EXECUTE SECOND HALF OF THE DATA
      TASK2ARY(1)=3
      TASK2ARY(3)='TASK 2'H
C
      CALL TSKSTART (TASK2ARY,PLLEL,DATA(20001),20000)
C
C        NOW WAIT FOR BOTH TO FINISH
      CALL TSKWAIT (TASK1ARY)
      CALL TSKWAIT (TASK2ARY)
C
C        AND PERFORM SOME POST-EXECUTION CLEANUP
      ...
      END
```

In this example, TSKSTART is called once for each of the two tasks. Alternatively, the second TSKSTART could be replaced by a call to PLLEL and the TSKWAIT removed. This alternative approach reduces the overhead of the additional task, but it can make understanding the program structure more difficult. The two approaches produce the same results.

5.1.7 TSKVALUE

TSKVALUE retrieves the user identifier (if any) specified in the task control array used to create the executing task.

Format:

```
CALL TSKVALUE (return)
```

return Value held in word 3 of the task control array. A 0 is returned if the array length was less than 3 or if the task is the initial, root task (described in subsection 5.1.1, Tasks).

Example:

```
      SUBROUTINE PLLEL (DATA, SIZE)
      INTEGER SIZE
      REAL DATA(SIZE)
C
C        DETERMINE WHICH OUTPUT FILE TO USE
      CALL TSKVALUE(IVALUE)
      IF (IVALUE .EQ. 'TASK 1'H) THEN
        IUNITNO = 3
      ELSEIF (IVALUE .EQ. 'TASK 2'H) THEN
        IUNITNO = 4
      ELSE
        CALL ABORT          ! Error condition; do not continue.
      ENDIF
      ...
      END
```

In this example, the TSKVALUE call identifies which task is presently executing subroutine PLLEL.

5.1.8 TSKLIST

TSKLIST lists the status of each existing task, indicating whether the task is running, ready to run, or waiting. If the task is waiting, TSKLIST reports the address of the lock or event or the identifier of the task waited upon.

Format:

```
| CALL TSKLIST (dn) |
```

dn Optional name or unit number of the dataset receiving the task status list; the COS default is $OUT, and the UNICOS default is the standard output file, **stdout**.

5.2 SCOPES AND PROTECTION

This subsection discusses CFT77 and CFT implementations of COMMON and TASK COMMON, private data, and locks.

5.2.1 SHARED DATA

Multitasked programs written in Fortran should keep shared data in COMMON blocks. Space for such data is statically allocated at load time, and the variable contents are available for the life of the program. When a subroutine is called, any COMMON variables maintained in registers are stored into memory before the call. (Local data can remain in registers.)

****

CAUTION

You should not pass COMMON variables as parameters between subroutines in a multitasking system. These variables, even if they are located in COMMON blocks in a calling subroutine, are not treated as COMMON variables in the called subroutine, and the compiler may perform optimizations that lead to unexpected problems.

****

5.2.2 TASK COMMON DATA

Data that is global between subroutines, but private to a task, should be kept in TASK COMMON blocks. TASK COMMON is a Cray Fortran language extension that operates to ensure that, when a program unit with TASK COMMON blocks is compiled, each task has its own copy of each TASK COMMON block used in the task. These blocks are set up when the task begins and are released when the task completes. Be sure to specify stack storage when using TASK COMMON (see subsection 5.2.3, Private Data); with static allocation, TASK COMMON blocks are treated the same as regular COMMON blocks. The format for a TASK COMMON declaration in Fortran is as follows:

```
TASK COMMON /cbname/ nlist
```

cbname Task common block name. TASK COMMON must be named.

nlist List of variable names, array names, and array declarators, separated by commas.

Variables in TASK COMMON blocks cannot be used in SAVE or DATA statements or with NAMELIST I/O. Except for these restrictions, these variables can be used in the same way as any other variables declared in COMMON.

5.2.3 PRIVATE DATA

Multitasked programs written in Fortran should keep private data in a stack. You can accomplish this by compiling these programs with the following parameter on the CFT or CFT77 control statement:

```
CFT,ALLOC=STACK, ...
CFT77,ALLOC=STACK, ...
```

or the corresponding UNICOS command line:

```
cft  -a stack ...
cft77  -a stack ...
```

When compiling in this mode, the compiler allocates and accesses private variables from a stack frame specific to the subroutine invocation. In this manner, private variables are truly private to the task executing the subroutine; they do not conflict with any other task executing the subroutine. (Not all private variables will necessarily reside on the stack; the compiler may choose to use registers during the subroutine and never store the variable in memory. This is especially true for CFT77.)

****

CAUTION

Using SAVE or DATA statements causes the referenced variables to be assigned to static memory locations, regardless of the setting of the ALLOC parameter. CFT77 and CFT generate a warning message for each variable in a DATA statement that is not also in a SAVE statement if stack allocation is in effect.

****

Ensure that CRI and user libraries used in multitasking mode have been compiled or assembled to use a stack. The CRI site analyst handles the CRI libraries, but you are responsible for correctly compiling or assembling your own libraries.

Stack mode introduces some additional overhead to subroutine linkages. This overhead results from the need to allocate the stack space at run time. The stack management routines may be required to make expensive system requests to obtain more space. LDR and SEGLDR support several parameters that can reduce the possibility and number of such calls at the expense of obtaining memory space before it is needed. (See subsection 5.4, Tuning, for more information.)

Not all code needs to have its private variables allocated on a stack. For example, subroutines that are nonreentrant or serially reusable can be compiled with the STATIC option.

Code with statically allocated private variables can be combined with stack-allocated private variables in a user program, but you should do this carefully. If two tasks use a subroutine with statically allocated private variables at the same time, unpredictable results and intermittent errors can occur. You should completely compile and test a program with stack-allocated private variables before making the effort to introduce modules recompiled with statically allocated private variables.

5.2.4 LOCKS

Locks are the macrotasking facility for monitoring critical regions of code. The operation of locks follows the general description provided in section 2, Concepts.

Integer variables representing locks are called *lock variables*. Lock variables should be kept in COMMON blocks to ensure their residency in Central Memory across subroutine calls. Henceforth, the terms *lock* and *lock variable* are used interchangeably.

Example:

```
      PROGRAM MULTI
      INTEGER LKINPUT,LKOUTPUT,LKCALL
      REAL    INDATA(20000),OUTDATA(20000)
      COMMON  /CBINPUT/  LKINPUT,INDATA
      COMMON  /CBOUTPUT/ LKOUTPUT,OUTDATA
      COMMON  /MISC/     LKCALL
      ...
      END
```

In this example, two locks were placed in the same COMMON blocks as the data to which they correspond. This is not necessary, but it makes the program more understandable.

5.2.5 LOCKASGN

LOCKASGN identifies an integer variable that the program intends to use as a lock. You must call the LOCKASGN subroutine for each lock variable before the lock variable is used with any of the other lock subroutines. A lock is given an initial state of cleared or off.

The optional second argument is for use by library subroutines that require a lock variable internally. The argument ensures that a lock variable is assigned only the first time LOCKASGN is called for that lock. Hence, if multiple tasks call the subroutine, the different invocations do not reassign the same lock.

Format:

```
CALL LOCKASGN (name[,value])
```

name Integer variable to be used as a lock. The library stores an identifier into this variable. You must not modify a lock variable after the call to LOCKASGN until that variable is released by a call to LOCKREL.

value The initial integer value of the lock variable. LOCKASGN stores an identifier into a variable only if the variable still contains the value. If you do not specify *value*, an identifier is unconditionally stored into the variable.

Example:

```
PROGRAM MULTI
INTEGER LKINPUT,LKOUTPUT,LKCALL,ITID(2)
REAL    INDATA(20000),OUTDATA(20000)
COMMON  /CBINPUT/  LKINPUT,INDATA
COMMON  /CBOUTPUT/ LKOUTPUT,OUTDATA
COMMON  /MISC/     LKCALL
EXTERNAL SUB1
...
CALL LOCKASGN (LKINPUT)
CALL LOCKASGN (LKOUTPUT)
CALL LOCKASGN (LKCALL)
ITID(1) = 2
CALL TSKSTART(ITID,SUB1)
CALL SUB1
...
END
```

```
SUBROUTINE SUB1
COMMON /LOCK1/ LOCK1
DATA LOCK1 /-1/
...
CALL LOCKASGN (LOCK1,-1)
...
END
```

****

CAUTION

If a lock variable is not assigned before use in any of the other lock subroutines, the results are unpredictable. One common symptom is an abort with an ERROR EXIT and a P address of zero.

****

5.2.6 LOCKON

LOCKON sets a lock and returns control to the calling task. If the lock is already set, the task is suspended until the lock is cleared by another task. In either case, the task sets the lock when it next resumes execution of user code. This means that placing LOCKON before a critical region ensures that the code in that region is executed only when the task has unique access to the lock.

Format:

```
CALL LOCKON (name)
```

name Integer variable used as a lock

5.2.7 LOCKOFF

LOCKOFF clears a lock and returns control to the calling task. The act of clearing the lock may allow another task to resume execution, but this is transparent to the task calling LOCKOFF.

Format:

```
 ________________________
|                        |
| CALL LOCKOFF (name)    |
|________________________|
```

name Integer variable used as a lock

Example:

```
      PROGRAM MULTI
      INTEGER LKOUTPUT
      REAL    OUTDATA(20000)
      COMMON  /CBOUTPUT/ LKOUTPUT,OUTDATA
      ...
      CALL LOCKASGN (LKOUTPUT)
      ...
C
      CALL LOCKON (LKOUTPUT)
      DO 100 I=1,20000
        OUTDATA(I)=MAX(OUTDATA(I),0.0)
 100  CONTINUE
      CALL LOCKOFF (LKOUTPUT)
C
      ...
      END
```

5.2.8 LOCKREL

LOCKREL releases the identifier assigned to the lock. If a task is waiting for the lock, or if the lock is set, an error results. This subroutine is useful primarily for detecting errors that arise when a task waits on a lock that is never cleared. The lock variable can be reused following another call to LOCKASGN.

Format:

```
 ________________________
|                        |
| CALL LOCKREL (name)    |
|________________________|
```

name Integer variable used as a lock

Example:

```
      PROGRAM MULTI
      INTEGER LKOUTPUT
      REAL    INDATA(20000),OUTDATA(20000)
      COMMON  /CBOUTPUT/ LKOUTPUT,OUTDATA
      ...
      CALL LOCKASGN (LKOUTPUT)
      ...
C
      CALL LOCKON (LKOUTPUT)
      DO 100 I=1,20000
        OUTDATA(I)=MAX(OUTDATA(I),0.0)
 100  CONTINUE
      CALL LOCKOFF (LKOUTPUT)
C
      ...
      CALL LOCKREL (LKOUTPUT)
C
      END
```

5.3 SYNCHRONIZATION

Two types of synchronization mechanisms are available with macrotasking: events and barriers. An event is an explicit signaling device. Barriers support a more implicit join construct for tasks.

5.3.1 EVENTS

This subsection discusses the subroutines that support events. Events allow signaling between tasks and have two states: cleared and posted. If an event is posted, one task is telling another that a certain action has been accomplished or a certain point in the program has been reached. If the action has not been accomplished or the point reached, the event remains in the cleared state.

An integer variable represents an event. Keep event variables in COMMON blocks to ensure their residency in memory across subroutine calls. Henceforth, the terms *event* and *event variable* are used interchangeably. The following example shows sample declarations:

Example:

```
PROGRAM MULTI
INTEGER EVSTART,EVDONE
COMMON  /EVENTS/ EVSTART,EVDONE
...
END
```

5.3.1.1 EVASGN

EVASGN identifies an integer variable that the program intends to use as an event. You must call this subroutine for an event variable before that variable is used with any of the other event subroutines. The initial state of the event is cleared. A data statement can initialize the event to the value in the optional argument so that the event can be assigned in a task. The first call assigns the event; further calls are ignored.

The optional second argument is for use by library subroutines that require an event variable internally. The argument ensures that an event variable is assigned only the first time EVASGN is called for that event. Hence, if multiple tasks call the subroutine, the different invocations will not reassign the same event.

Format:

```
CALL EVASGN (name[,value])
```

name Integer variable to be used as an event. The library stores an identifier into this variable. Do not modify the variable after the call to EVASGN unless a call to EVREL first releases the variable.

value The initial integer value of the event variable. EVASGN stores an identifier into a variable only if that variable still contains the value. If you do not specify *value*, an identifier is unconditionally stored into the variable.

Example:

```
      PROGRAM MULTI
      INTEGER EVSTART,EVDONE
      COMMON /EVENTS/ EVSTART,EVDONE
      ...
      CALL EVASGN (EVSTART)
      CALL EVASGN (EVDONE)
      ...
      END


      SUBROUTINE SUB1
      INTEGER EVENT1
      COMMON /EVENT1/ EVENT1
      DATA EVENT1 /-1/
      ...
      CALL EVASGN (EVENT1,-1)
C     ...
      END
```

****

CAUTION

If an event variable is not assigned before it is used in any of the other event subroutines, the results are unpredictable. One common symptom is an abort with an ERROR EXIT and a P address of zero.

****

5.3.1.2 EVWAIT

EVWAIT waits until the specified event is posted. If the event is already posted, the task resumes execution without waiting. EVWAIT does not change the state of the event.

Format:

```
| CALL EVWAIT (name) |
```

name Integer variable used as an event

Example:

```
SUBROUTINE MULTI2
INTEGER EVSTART,EVDONE
COMMON /EVENTS/ EVSTART,EVDONE
...
CALL EVWAIT (EVSTART)
...
END
```

In this example, EVWAIT tests the event EVSTART to see whether it is posted. If it is posted, execution continues; if it is not, execution halts until some other task posts the event.

5.3.1.3 EVPOST

EVPOST posts an event and returns control to the calling task. Posting an event allows all other tasks waiting on that event to resume execution, but this condition is transparent to the task calling EVPOST. Posting an already posted event has no effect (posts are not queued).

Format:

```
CALL EVPOST (name)
```

name Integer variable used as an event

Example:

```
PROGRAM MULTI
INTEGER EVSTART,EVDONE
COMMON /EVENTS/ EVSTART,EVDONE
...
CALL EVASGN (EVSTART)
CALL EVASGN (EVDONE)
...
CALL EVPOST (EVSTART)
END
```

5.3.1.4 EVCLEAR

EVCLEAR clears an event and returns control to the calling task. Tasks subsequently performing EVWAIT calls on that event must wait. If the event is not cleared, the posted condition remains. When a single event post is required (a simple signal), call EVCLEAR immediately after EVWAIT to indicate that the posting of the event has been detected.

Format:

```
CALL EVCLEAR (name)
```

name Integer variable used as an event

Example:

```
SUBROUTINE MULTI2
INTEGER EVSTART,EVDONE
COMMON /EVENTS/ EVSTART,EVDONE
...
CALL EVWAIT (EVSTART)
CALL EVCLEAR (EVSTART)
...
END
```

5.3.1.5 EVREL

EVREL releases the identifier assigned to an event. If a task is waiting for the event, an error results. This subroutine is useful primarily in detecting erroneous uses of an event outside the region the program has planned for it. The event variable can be reused following another call to EVASGN.

Format:

```
CALL EVREL (name)
```

name Integer variable used as an event

Example:

```
      PROGRAM MULTI
      INTEGER EVSTART,EVDONE
      COMMON  /EVENTS/ EVSTART,EVDONE
      ...
      CALL EVASGN (EVSTART)
      CALL EVASGN (EVDONE)
      ...
      CALL EVPOST (EVSTART)
      ...
C        EVSTART WILL NOT BE USED FROM NOW ON
      CALL EVREL (EVSTART)
      ...
      END
```

5.3.2 BARRIERS†

A barrier is a synchronization point in an application beyond which no task will proceed until a specified, predetermined number of tasks have reached the barrier. Three routines, BARASGN, BARSYNC, and BARREL, perform barrier synchronization.

5.3.2.1 BARASGN

BARASGN identifies an integer variable for the program to use as a barrier. You must call this subroutine for a barrier variable before that variable is used with any of the other barrier subroutines.

The initial state of the barrier is closed. A barrier remains closed until its count is met; that is, the BARSYNC routine has been called with this variable by the appropriate number of tasks. At this point, all waiting tasks are allowed to execute and the barrier is once again closed.

Format:

```
CALL BARASGN (name, value)
```

name Integer variable to be used as a barrier. The library stores an identifier into this variable. Do not modify the variable after the call to BARASGN unless a call to BARREL first releases the variable.

† Deferred COS implementation

value The integer number of tasks, between 1 and 31 inclusive, that must call BARSYNC with *name* before the barrier is opened and the waiting tasks are allowed to proceed.

5.3.2.2 BARSYNC

BARSYNC registers the arrival of a task at a barrier. This causes the barrier's count to be decremented by 1. If the new count is greater than 0, the task waits. If the new count is 0, the task is permitted to proceed through the barrier, all tasks waiting at the barrier are permitted to resume execution, and the barrier is closed, with the count reset to the initial value set with the BARASGN call.

Format:

```
 ________________________
|                        |
| CALL BARSYNC (name)    |
|________________________|
```

name Integer variable used as a barrier

5.3.2.3 BARREL

BARREL releases the identifier assigned to a barrier. If a task is waiting for passage through the barrier, an error results. This subroutine is useful primarily in detecting erroneous uses of a barrier outside the region the program has planned for it. The barrier variable can be reused following another call to BARASGN.

Format:

```
 ________________________
|                        |
| CALL BARREL (name)     |
|________________________|
```

name Integer variable used as a barrier

5.3.2.4 Example of using barriers

The following example shows a simple use of the barrier synchronization mechanism. The barrier variable, LOOPBAR, is assigned a count of 8 and eight tasks are started. The tasks wait at the synchronization point until all eight reach the beginning of the 200 loop for each iteration of that loop.

```
      PROGRAM MULTI
      COMMON / SYNCCOM/ LOOPBAR
      INTEGER LOOPBAR
      INTEGER I, TASKINFO(2,8)
      EXTERNAL BARSUB
C
      CALL BARASGN(LOOPBAR, 8)
C
      DO 100 I = 1, 8
        TASKINFOR(1,I) = 2
        CALL TSKSTART(TASKINFO(1,I), BARSUB)
100   CONTINUE
      DO 110 I = 8, 1, -1
        CALL TSKWAIT(TASKINFO(1,I))
110   CONTINUE
      CALL BARREL(LOOPBAR)
C     ...
      END

      SUBROUTINE BARSUB
      COMMON / SYNCCOM/ LOOPBAR
      INTEGER LOOPBAR
      INTEGER I, J
C
      DO 300 I = 1, 100
        CALL BARSYNC(LOOPBAR)
        DO 200 J = 1, 500
C       ...
200       CONTINUE
300   CONTINUE
C
      RETURN
      END
```

5.4 TUNING

The multitasking system software design lets you perform tuning without the need to rebuild libraries or other system software. A library routine, TSKTUNE, performs the tuning.

5.4.1 TSKTUNE

TSKTUNE modifies tuning parameters within the library scheduler, which manages and schedules tasks within programs. (The design of multitasking software is described in appendix F.) Each parameter has a default setting within the library and can be modified at any time to another valid setting.

For the most predictable results, you should call TSKTUNE during application initialization or when there is only one task active.

This routine should not be used for multitasking on a CRAY-1 computer system. The parameters are not relevant in a single-CPU environment.

The effects of this routine may not be measurable in a batch environment because of variable conditions between and during runs.

Format:

```
CALL TSKTUNE (keyword1, value1, keyword2, value2, ...)
```

keyword$_i$ An ASCII character string, as follows:

'DBACTIVE' Deadband for activation of logical CPUs. (Logical CPUs are described in appendix F.1.1.) This is the number of additional user tasks that can be readied for execution before an additional logical CPU is activated or acquired. This allows a queue of tasks to be built before another CPU is acquired to process these tasks. For example, if DBACTIVE is set to 2, the scheduler will activate a suspended logical CPU if more than two tasks are ready to execute. The value of DBACTIVE may range from 0 to the largest integer value (the number of logical CPUs is equal to the number of user tasks limited by MAXCPU). The initial value is 0.

'DBRELEAS' Deadband for release of logical CPUs. If more logical CPUs are allocated to the job than there are tasks, the deadband reflects the maximum number retained. Any in excess of this number are released to the system, requiring an exchange to the operating system. The initial value is set to 1 less than the number of physical CPUs available on the system or to 1, whichever is greater. (For example, DBRELEAS is set to 3 for a four-processor CRAY X-MP computer system and to 1 for a single-processor CRAY X-MP computer system.) Setting DBRELEAS to less than this value may cause an excessive number of CPUs to be deleted and acquired and a correspondingly long list of CPUs in the log file. The value

'DBRELEAS' (continued) of DBRELEAS can range from 0 (representing immediate return) to the value of MAXCPU.

'HOLDTIME' Number of clock periods (CPs) to hold a logical CPU while waiting for tasks to become ready and before releasing the CPU to the operating system. This parameter lets a user hold additional logical CPUs in a job when executing a nonmultitasked section of code and have these CPUs quickly available when the program reenters multitasking mode. The value of HOLDTIME may range from 0 (return logical CPUs immediately to the operating system) to the largest integer value. The initial value is 100,000 CPs. This parameter is ignored when microtasking is running.

'MAXCPU' Maximum number of logical CPUs allowed for the job. The initial value is set to the number of physical CPUs available on the system. (For example, MAXCPU is set to 2 for a dual-processor CRAY X-MP computer system.) The value of MAXCPU can range from 1 to the value of an installation parameter that can vary from site to site; the parameter limits the number of tasks in the system. Under COS, this is the parameter I@MAXNUT. Under UNICOS, the parameter is MAXUP, and its value is a site-specific constant minus the number of other processes the user has running at the current time. (Thus, users are limited to a finite number of simultaneous processes.)

'SAMPLE' Number of CPs between checks of the ready queue. This parameter is used with the HOLDTIME parameter. SAMPLE adjusts the frequency of sampling the ready queue (containing tasks ready to execute) when a logical CPU is waiting for a task to become ready. If the ready queue is sampled too often, excessive memory contention may result. The value of SAMPLE may range from 0 (sample the ready queue as often as possible) to the largest integer value (the ready queue is effectively never sampled and ready tasks are never executed by the waiting processor). The initial value is 500 CPs. This parameter is ignored when microtasking is running.

value$_i$ An integer

Specify the parameters in keyword-value pairs, though they need not be in any particular order.

NOTE

TSKTUNE does not check to ensure that a value passed to it is within its specified range.

In general, the settings in a dedicated environment should be different from those in a batch environment. In a dedicated environment, the deadbands and the loop counts can be high, but generally the cost of these wasted CPU cycles is less than the cost of constantly returning to the operating system to change the number of tasks. In a batch environment, the deadbands and loop count should be low (the default settings) because the wasted CPU time could degrade total system throughput. In either case, the operating system accounting charges the job for any time spent idling in an unused CPU.

In a dedicated environment, set the value of MAXCPU to equal either the number of physical CPUs or the number of physical CPUs plus 1. The latter can improve performance in a design with one task performing I/O and the remaining tasks performing computations. In a batch environment, keep the value low, though it is of less consequence. Generally, set the value of DBRELEAS to equal the value of MAXCPU minus 1, so that unused logical CPUs are not returned to the system too quickly.

Examples:

```
CALL TSKTUNE ('DBACTIVE',1,'HOLDTIME',0)
CALL TSKTUNE ('MAXCPU',1)
```

The first example keeps one more user task than there are logical CPUs and cuts back to one logical CPU as quickly as possible. The second limits the job to a single logical CPU.

5.4.2 LDR AND SEGLDR MEMORY MANAGEMENT TUNINGS†

The loader statements LDR and SEGLDR include parameters and directives dealing with memory stacks and heaps. These parameters are described in the COS Version 1 Reference Manual for LDR and in the Segment Loader (SEGLDR) Reference Manual for SEGLDR. Table 5-1 summarizes the options.

† LDR is available only under COS.

Table 5-1. Summary of Loader Options

Function	LDR Parameter	SEGLDR Directive
Define initial stack size and increment	STK	STACK
Define initial heap size and increment	MM	HEAP
Define minimum size of a free heap block	MMEPS	HEAP

If you do not specify values for the heap initial size and increment at load time, default values are used. If the STACK option is not specified at compile time, default values are used for the stack. If you use multitasking with CAL programs that do not include reentrant Fortran code, you must specify at load time that a stack is needed.

The following examples show settings that should provide enough space for a multitasked program with a few tasks and a moderate depth of subroutine use. The examples include increments in case of stack overflow. Optimal initial settings, however, depend upon the specific job and application.

LDR parameters:

```
LDR,MM=15000:5000,STK=5000:1000,...
```

SEGLDR directives:

```
HEAP=15000+5000
STACK=5000+1000
```

While developing and debugging multitasked code, determine, if possible, optimal settings for the initial and increment settings. If space is not a concern, slight performance improvements can be gained by determining the maximum required space for the stack and heap. Doing so may not be easy because requirements can change from run to run if the execution sequences of the tasks change.

DEBUG gives both heap and stack statistics if a DUMPJOB statement is executed immediately after the multitasked program.† (See section 7, Debugging, for sample output from DEBUG.) The values written by DEBUG are useful only if the initial stack and heap sizes are smaller than what is required; this is the case because the DEBUG values reflect the actual sizes, which may be more than what was needed. In addition, the library subprograms IHPSTAT† and HPDUMP† can be called from a user program to obtain current heap statistics.

If you set the parameters to meet maximum storage requirements, allocation of memory space occurs at load time rather than at run time. Some slack can be built into the initial request, or the increment setting can be set to catch cases requiring additional space. If the increment is set to zero, an attempted expansion or overflow aborts the job.

The size of the minimum heap block is not likely to have a noticeable effect on performance. It prevents over-fragmentation of the heap manager's free space queue, which could lengthen search times for new blocks. This would be a problem only if a large amount of dynamic and varying use is made of the heap manager.

NOTE

The use of multitasking with either LDR overlays or SEGLDR segments is not specifically supported. (See subsection 5.5.2, Overlays and Segments.) The parameters described here apply to normal loads with LDR or nonsegmented loads with SEGLDR.

5.5 UNDERLYING ASSUMPTIONS

This subsection covers various assumptions and warnings about the user program and the environment in which it is used.

5.5.1 COS AND UNICOS ASSUMPTIONS

The macrotasking implementation is available with COS and UNICOS. The following paragraphs describe some assumptions that apply to one or both operating systems. You may have to consult with your system administrator or a CRI site analyst on one or more of these concerns.

---

† Deferred UNICOS implementation

The libraries used with multitasked programs must have been created with the multitasking assembly option through the use of the multitasking version of the system text dataset ($SYSTXT for CAL Version 1, $SYSDEF for CAL Version 2 under COS, and **asdef** for CAL Version 2 under UNICOS). The multitasking assembly option generates the libraries with the following features:

- Stack option of the calling sequence
- Nonreentrant I/O subroutines in $IOLIB protected with locks
- Multitasking library subroutines enabled for multiple tasks

The default libraries at most sites are built with the static option of the calling sequence and should not be used with multitasked programs.

On a CRAY X-MP computer system, COS should be configured to allow multiple tasks for a job. For a CRAY-1 computer system, COS should limit a job to a single task.

UNICOS on a CRAY X-MP computer system allows the same number of tasks in a program as the number of simultaneous processes it allows in a process-group. On a CRAY-1 computer system, UNICOS disallows the creation of any additional tasks.

5.5.2 OVERLAYS AND SEGMENTS

The multitasking features provide no explicit support for LDR overlays or for SEGLDR segments. Nothing prevents you from using them together, but the serial nature of loading segments and overlays conflicts with the parallel nature of multitasking. You must take care if attempting a combination of multitasking and overlays or segments.

You can use multitasking with segmented loads if the set of segments in memory is not changed during the course of the multitasked stage of the job. Keeping the subroutines used for multitasking completely within the root or some segment is the safest way to ensure that memory does not change. SEGLDR currently cannot verify or enforce this. Overlays can be used under the same limitations.

5.5.3 EXTENDING BLANK COMMON

Some Fortran programs use a nonstandard technique known as extending blank COMMON to provide dynamic memory management within user space. The use of stacks, though, sometimes requires expansion of memory in user space (see subsection 5.4.2, LDR and SEGLDR Memory Management Tunings). Only one of these two memory extension mechanisms can be in control of a job.

If it is necessary to extend blank COMMON, set the stack area to a fixed size (increment of 0) and place it below blank COMMON. Attempts to expand the stack then cause a job to abort.

If blank COMMON does not need to be expanded, the stack can be placed above it and allowed to expand. Attempts to then expand blank COMMON cause unpredictable results. LDR and SEGLDR parameters let you specify the proper configuration.

As an alternative to extending blank COMMON, consider using user-callable heap allocation routines (such as HPALLOC and HPDEALLC). The stack management routines use these heap allocation routines, allowing both stacks and user code to share the same memory area. The Programmer's Library Reference Manual describes these subroutines.

5.5.4 CFT77 AND CFT OPTIMIZATION

The CFT77 and CFT compilers generate heavily optimized code. These optimizations are usually transparent in a program run in nonmultitasked mode, but in a multitasked application they can cause problems. The following describes three such optimizations.

When writing programs, programmers have a mental model of the way in which they expect the code to be executed. All models are simplifications, and with a nonmultitasked application, the simplifications in the model do not cause problems. When the model is multitasked, however, the deficiencies begin to appear. The following problems relate to these simplifications:

Problem: Reluctance to store variables to memory. When writing Fortran code, most programmers mentally use a model in which variables are stored to or loaded from memory at each reference. But because memory loads and stores are expensive, compilers try to keep variables in registers (A, B, S, T, or V) for as long as possible. CFT77 goes a step further; it may determine that it is unnecessary to ever store a variable to memory. This could be a problem if another task is trying to use the same variable; the optimizations could result in the other task using an old value.

Solutions: Use variables in COMMON for shared data and protect them with locks. Both CFT and CFT77 store such variables to memory on a subroutine call. Use the SAVE statement for local variables that may be shared with argument lists.

Problem: Reordering statements. Another programming model assumes that statements are executed precisely in the order listed. But compilers are free to reorder statements and parts of statements as long as the data dependencies between statements are not changed. If two successive statements appear to be unrelated, the compiler may reorder them. With multitasking, some statements (such as CALL TSKSTART) have significant side effects, of which the compiler may not be aware.

Solutions: The solutions are similar to those for the previous problem. The compiler sees locks in the same way it sees any other subroutine, so you must make sure locked data is kept between a LOCKON-LOCKOFF pair. Putting it in COMMON is one way to ensure this.

Problem: Temporary storage. Another programming model sees expressions (such as N*M) and simple variables (such as N) as being interchangeable. This model ignores the fact that to pass an expression as an argument, the compiler must compute the value, obtain temporary storage space, and store the computed value. This space is obtained from the stack, and the compiler can reuse the space when it appears to be no longer needed. This is a problem when the address of this space is passed to another task, but the space is reused before the new task receives it.

Solutions: Compute expressions explicitly, and assign the result to a variable. Then pass that variable to the other task.

The following code segment reflects some of these problems. Assume that the code that called MYSUB had LOCKI and I safely stored in COMMON but decided to pass them as arguments to MYSUB because of the software design. Assume further that actual code is being executed within the DO loop, but none of the code uses I except the following:

```
      SUBROUTINE MYSUB (LOCKI,I)
      EXTERNAL NEWSUB
      INTEGER ITCA(2)
C
      ITCA(1)=2
C
      DO 10 II=1,10
        CALL LOCKON (LOCKI)
          I=I+1
        CALL LOCKOFF (LOCKI)
C
  10  CONTINUE
C
      CALL TSKSTART (ITCA,NEWSUB,3*I)
      CALL OTHERSUB (4*I)
C
      END
```

With CFT, the generated code executes as follows. (The code generated by CFT77 is similar.)

```
      SUBROUTINE MYSUB (LOCKI,I)
      EXTERNAL NEWSUB
      INTEGER ITCA(2)    (allocated on stack)
C
      ITCA(1)=2
C
        (reg T00 = 1)
      DO 10 II=1,10
        CALL LOCKON (LOCKI)
        (reg T00 = reg T00+1)
        CALL LOCKOFF (LOCKI)
C
  10  CONTINUE
        (I = reg T00)
C
        (compute 3*I and put on stack)
      CALL TSKSTART (ITCA,NEWSUB,3*I)
        (compute 4*I and put on stack on top of 3*I)
      CALL OTHERSUB (4*I)
C
      END
        (and possibly complete task execution)
```

What problems have resulted here?

- The LOCKON-LOCKOFF pair have not protected the store to memory of I. They have just protected the update of register T00 (local to each task). The store to I occurs after the LOCKOFF and is not protected. This can be fixed by putting I in a COMMON declaration in MYSUB rather than having it as an argument.

- The expression 3*I, passed to TSKSTART, is quickly replaced by 4*I; this will probably happen before the new task is created and reads the value. This can be fixed by assigning 3*I to a new variable in a COMMON block and passing that variable to TSKSTART.

The preceding code looked good and would work correctly in a single-tasked environment, but it could fail in a multitasked environment. By being aware of compiler optimizations, you can avoid these problems.

5.5.5 COS REPRIEVE PROCESSING

Multitasked programs can use COS reprieve processing, but you must be careful to consider the two types of reprieve conditions: user caused and environmental.

For a user-caused condition (such as an operand range error or a floating-point error), each task that can cause the error must issue a SETRPV request for the condition or conditions of interest. When a condition occurs, the reprieve code receives control from the task that caused the error, and any other tasks are suspended.

For an environmental condition (such as interactive attention), each task that can be executing when the condition arises must issue a SETRPV request for the conditions of interest. When a condition occurs, the reprieve code receives control from one of the running tasks; any other tasks are suspended. If one task omits the SETRPV and happens to be executing when the condition occurs, the error condition is not reprieved and the job aborts.

In the case of either a user-caused or an environmental error, the reprieve code should execute either a CONTRPV or an ENDRPV request upon completion of reprieve processing. If CONTRPV is executed, all tasks are resumed, if possible; if ENDRPV is executed, termination of the job step completes.

Reprieve processing currently works correctly only for a program that has the same number of tasks as logical CPUs, because reprieves are enabled for logical CPUs rather than for tasks.

Each task must have its own Exchange Package save area to be used during a reprieve. This area must not reside on a stack for a task that can be completed before the address is used. Put it in COMMON, or use the SAVE statement.

5.5.6 COS IOAREA LOCK

The COS IOAREA control statement and macro do not work with a multitasked job. Their purpose is to prevent the user from accessing the portion of the user field following the high limit of memory address (HLM) where the Dataset Parameter Area (DSP) and I/O buffers normally reside. In a multitasked job, the DSPs and buffers are allocated from the heap, before HLM, and are thus not protected by the IOAREA lock.

5.5.7 NONREENTRANT LIBRARY ROUTINES

Most of the routines in the run-time libraries either are reentrant or have internal locks to ensure that they are single threaded. Some library routines, however, must be locked at the user level if they are used by more than one task.

Random access dataset I/O routines (READMS/WRITMS, READDR/WRITDR, and GETWA/PUTWA) are not internally locked. You must lock each call to these routines if they are called from more than one task.

The sort/merge routines† are not reentrant and are not locked. An entire series of calls to the sort/merge routines must be locked, from SAMSORT or SAMMERGE to SAMGO. Each routine in the group uses data built up in previous calls, so locking each individual call is not sufficient.

The COS table manager routines extend blank COMMON. Table manager routines are reentrant, but references to the COMMON block arrays used to store the lengths and locations of tables are not locked. If the table manager is used with a multitasked program, calls to the table manager should be locked and the heap must be placed before blank COMMON. After doing so, attempts to expand stacks will abort the job. The UNICOS table manager routines use the heap manager, which permits their use in a multitasked program.

Debugging routines DUMP†, PDUMP†, and CRAYDUMP† are all reentrant, but you should put locks around calls to these routines if they use the same datasets in different tasks so that the outputs from the tasks are not interspersed.

---

† Deferred UNICOS implementation

Routines TRBK, SNAP†, and DUMPJOB† are not locked and are not reentrant. If one of these routines is called from more than one task, locks should be placed around the calls.

You should lock calls to TIMEF.

5.6 MULTITASKING EXAMPLE

This subsection describes the overall design of a program that uses the multitasking features previously described. The next subsection expands this example.

5.6.1 GENERAL APPLICATION

The example program starts by reading data a unit at a time from an input dataset. Each unit could represent, for example, data recorded at a certain time, and the input dataset could be a sequence of such readings. Each unit of data is processed, and a corresponding output unit is written. The program repeats these operations until the input dataset is completely read.

5.6.2 INITIAL TASK

The main program, or initial task, performs the following operations:

1. Initializes control variables
2. Starts a task to write the output unit
3. Enters main loop and performs the following:
    a. Reads a unit of data (if end-of-data is reached, it leaves the loop)
    b. Performs preprocessing that cannot be multitasked
    c. Starts four tasks, each of which processes 25% of the data and uses the same code for execution
    d. Waits for the tasks to complete

† Deferred UNICOS implementation

e. Performs postprocessing that cannot be multitasked

f. Waits for the output task to write the previous unit of data (or, in the case of the first time through the loop, to initialize itself)

g. Signals the output task to write the next unit of data

4. Performs the next iteration of the loop

5. Waits for the output task to write the last unit of data

6. Signals the output task that all data has been supplied

5.6.3 OUTPUT TASK

The output task allows output of one set of data to occur in parallel with processing of the next set of data. The only synchronization necessary is to ensure that one set is written before the next set is supplied.

The output task performs the following operations:

1. Initializes control variables and opens the dataset

   a. Begins main loop

   b. Signals ready condition

   c. Waits for the initial task to supply data (if the signal indicates that all data has been supplied, it leaves the loop)

   d. Outputs data

2. Performs the next iteration of the loop

5.6.4 PROCESSING TASKS

The processing tasks are four copies of the same module, each processing 25% of the data. Four tasks, created by the initial task to process a different quarter of the data, share this single set of instructions.

The shared module has the following operations:

1. Initializes control variables

2. Processes data

5.6.5 COS JCL

The following is JCL that could be used to run the example as a COS job:

```
JOB,JN=MTEXAMP,...
ACCOUNT,...
MULTI.                                  Access multitasking/stack libraries
CFT77,ALLOC=STACK.                      Code in $IN
SEGLDR,CMD='HEAP=15000+5000;STACK=5000+1000',GO.
                                        Allow stack and heap to grow as
                                        needed
```

5.6.6 INITIAL TASK CODE

The code for the main task is as follows:

```
      PROGRAM EXAMPLE
C
      EXTERNAL OUTPUT,PROCESS
C
      INTEGER OUTTCA(2),PR1TCA(2),PR2TCA(2)
C
C     Force arrays and variables to static storage
C
      COMMON /TCAS/OUTTCA,PR1TCA,PR2TCA,START1, START2
C
      INTEGER DATAOUT,OUTDONE
      COMMON /EVENTS/DATAOUT,OUTDONE
      INTEGER DATALOC
      COMMON /OUTDATA/DATALOC
C
      INTEGER START1, START2
      REAL DATAFILE (100000)
      COMMON /DATASET/DATAFILE,START1,START2
      DATA START1,START2/1,50001/
C
C     ... (Open input dataset)
C
C           (Initialize control variables)
C
```

```
      CALL EVASGN (DATAOUT)
      CALL EVASGN (OUTDONE)
C
C     (Initialize a task control array)
C
      OUTTCA(1)=2
      PR1TCA(1)=2
      PR2TCA(1)=2
C
C           Start a task to write the output unit
C
      CALL TSKSTART (OUTTCA,OUTPUT)
C
C           Enter main loop
C
 100  CONTINUE
C
C           Read unit of data (if end of data is reached, leave the loop)
C
      ... (read from input dataset to DATAFILE, END=1000)
C
C           Perform preprocessing that cannot be multitasked
C
C
C           Start two tasks, each processing 50% of the data
C
      CALL TSKSTART (PR1TCA,PROCESS,START1)
      CALL TSKSTART (PR2TCA,PROCESS,START2)
C
C           Wait for the tasks to complete
C
      CALL TSKWAIT (PR1TCA)
      CALL TSKWAIT (PR2TCA)
C
C           Perform postprocessing that cannot be multitasked
C
C           Wait for the output task to finish previous I/O
C
      CALL EVWAIT (OUTDONE)
      CALL EVCLEAR (OUTDONE)
C
C           Signal the output task to write the next unit of data
C
      DATALOC = ...
      CALL EVPOST (DATAOUT)
C
C           Perform next iteration of the loop
C
      GOTO 100
C
1000  CONTINUE
```

```
C
C           Wait for the output task to write previous unit of data
C
      CALL EVWAIT (OUTDONE)
C
C           Signal the output task that all data has been supplied
C
      DATALOC = - 99999
      CALL EVPOST (DATAOUT)
C
      END
```

5.6.7 OUTPUT TASK CODE

The following task performs the output operations:

```
      SUBROUTINE OUTPUT
C
      INTEGER DATAOUT,OUTDONE
      COMMON /EVENTS/DATAOUT,OUTDONE
C
      INTEGER DATALOC
      COMMON /OUTDATA/DATALOC
C
C           Initialize control variables, open the dataset
C
C           Begin main loop
C
 100  CONTINUE
C
C           Signal ready condition
C
      CALL EVPOST (OUTDONE)
C
C           Wait for the initial task to supply data
C
      CALL EVWAIT (DATAOUT)
      IF (DATALOC .LT. 0) GOTO 1000
      CALL EVCLEAR (DATAOUT)
C
C           Output data (based on DATALOC)
C
C           Perform next iteration of loop
C
      GOTO 100
C
1000  CONTINUE
C
      RETURN
      END
C
```

5.6.8 PROCESSING TASK CODE

The following task performs the actual data processing:

```
      SUBROUTINE PROCESS (IPTR)
C
      REAL DATAFILE (100000)
      COMMON /DATASET/DATAFILE
C
C           Initialize control variables
C
C           Process data (based on IPTR)
C
      RETURN
      END
```

6. PROGRAM ANALYSIS AND CONVERSION

The job of analyzing and converting programs to be multitasked requires an understanding of the parallelism concepts presented in previous sections and a knowledge of the function and correct use of the multitasking library utilities. This section describes a procedure for finding parallelism in programs, analyzing the independence requirements, and finally writing the multitasked code.

The original program is assumed to be debugged and working correctly.

6.1 CONDITIONAL MULTITASKING

A powerful debugging technique that you should consider when analyzing a program is simply to set up the modified program so that the multitasking can easily be turned on or off. This technique is mentioned here because it is best implemented from the start rather than after you find errors. If unexpected results are produced, rerun the same code without multitasking to help identify whether or not the problem is related to multitasking. The alternative is to try to maintain the nonmultitasked (original) program, but this is often difficult.

You might consider the following when implementing conditional multitasking with macrotasking:

- Define a COMMON logical variable that is set with input or with a data statement. This variable controls multitasking, and calls to the multitasking library subroutines should be made only if this variable has a value of TRUE.

- Set up a mechanism to base the partitioning of the data on the use of multitasking. If multitasking is turned off, for example, a task must process all of the data, not just a fraction of it. The program design dictates how this is best done.

One of the advantages of microtasking is that you can accomplish this technique easily by bypassing PREMULT and going directly to the compile step. This approach does not require anything special within the source code.

6.2 MOVING FROM STATIC TO STACK ENVIRONMENT

The first step in converting a program for multitasking is to verify the correct execution of the program in a stack environment. Codes that ran in a static environment may not execute properly in a stack environment because of changes in handling local variables.

In a static environment, the local variables in a program unit occupy fixed memory locations for the life of the program. This allocation allows these variables to be initialized at load time and permits their values to persist from one call of a subroutine to the next. While the ANSI Fortran standard calls for local variables to become undefined upon exiting the routine, many compilers (including all versions of CFT, if the BTREG option is not selected) allow this value retention across calls. Programs that rely on this feature are nonstandard.

In a stack environment, the local variables of a program unit occupy memory locations on a stack. The location of the stack space for a program unit depends on when the routine was called during execution. The dynamic nature of the stack mechanism precludes the handling of local variables in the same way as in the static environment. Local variables cannot be initialized at load time because their location is not known until run time. Values cannot be carried from one call to the next, because the stack space for each call may reside in a different memory location.

The following list describes some guidelines and situations you must be aware of when moving a program from a static to a stack environment.

- Local variables in a subroutine are not predefined, or they retain their values across repetitive calls.

  Example:

```
      SUBROUTINE WORK( B )
      INTEGER INIT
      IF( INIT.NE.0 ) GO TO 10
      A = 0.0
10    CONTINUE
      INIT = 1
      B = A
      RETURN
      END
```

The value of INIT is not initialized to any particular value and may not be zero on the first call to WORK. The value of INIT is not retained from one call to the next. Likewise, the value of A is not retained, and B may be assigned an arbitrary value on subsequent calls to WORK. Using the PRESET=INDEF directive with SEGLDR or the SET=INDEF option with LDR helps to identify static variables that were assumed to be initially defined as zero. The CFT INDEF option performs a similar function for stack variables. The CFT77 INDEF option performs a similar function for all local variables, static and stack.

- Local variables in DATA statements are allocated static storage locations. Using DATA statements to initialize local variables causes the compiler to change the allocation of these variables from stack to static, allowing initialization to be done at load time.

  Example:

```
      SUBROUTINE X ( B )
      LOGICAL FIRST
      DATA FIRST / .TRUE. /
      IF ( FIRST ) THEN
      FIRST = .FALSE.
      B = 1.0
      END IF
      B = 0.0
      RETURN
      END
```

  The variable FIRST is statically allocated in a stack compilation. This could cause a problem if several tasks test FIRST before any task changes it to FALSE.

- Library and Cray Assembly Language (CAL) routines called from separate tasks must be reentrant. Most scientific library software is reentrant or protected by system semaphores; a notable exception is RANF().

  Example:

```
      SUBROUTINE TASK( A )
      DIMENSION A(100)
      DO 10 I = 1, 100
        A(I) = RANF()
   10 CONTINUE
      RETURN
      END
```

It cannot be guaranteed that each task will get a distinct set of random number values.

- You must be attentive to passing local variables as arguments.

Example:

```
      PROGRAM MAIN
      DO 10 I = 1, NCPUS-1
        CALL TSKSTART( ID(1,I), TASK, I, 3*I )
        CALL SUB( 4*I )
   10 CONTINUE
      STOP
      END
```

The main program may change the value of local variable I before the tasks are able to read their arguments. The expression 3*I is evaluated into a temporary location whose address is passed to the tasks. This location may be reused in the evaluation of 4*I before the tasks are able to read their arguments. Passing arguments to TSKSTART has been responsible for a majority of previously encountered macrotasking bugs and should be avoided whenever possible.

- Avoid conflicts with compiler memory optimizations. CFT and CFT77 assume that variables not in COMMON blocks are unrelated to subroutines that do not contain the variable as an argument. Such variables need not be stored to memory before the call nor loaded from memory after the call, but rather they may be retained in a register.

Example:

```
      SUBROUTINE TASK( I )
      COMMON /LOCKS/LOCKI
      CALL LOCKON( LOCKI )
      I = I + 2
      CALL LOCKOFF( LOCKI )
      RETURN
      END
```

Because the variable I is not in a COMMON block and not in the argument list of either LOCKON or LOCKOFF, the compiler feels free to maintain the value of I in a register and to schedule the loading and storing of I in memory independently of the calls to LOCKON and LOCKOFF. The code for references to I may look as follows:

```
reg0 = I                    ! load I from memory
CALL LOCKON( LOCKI )
reg0 = reg0 + 2             ! increment I in register
CALL LOCKOFF( LOCKI )
I = reg0                    ! store I to memory
```

Because the assignment to I is outside the locked area, the result of this code is that the lock does not protect access to the variable I. It is recommended that shared variables be kept in COMMON blocks.

6.3 THREE STEPS TO ANALYZING AND CONVERTING CODE FOR MULTITASKING

The following three subsections describe the three steps in the CRI approach to converting serial code to multitasked code. This approach is oriented toward the macrotasking feature. For suggestions about microtasking, see section 4.

The first step is analyzing and understanding the program flow to determine what portions of the code can be multitasked. Analyzing the program for data dependencies is the next step, and the final step is to write the code that converts the serial program to a multitasked program.

6.3.1 LOCATING POTENTIAL PARALLELISM

The goal of the first step is to understand the program flow in order to locate portions of the program that have the potential to be multitasked. The potential exists if loops (potential DOALLs) exist, and if the granularity of work is sufficient to consider multitasking.

The following stages constitute this step:

Stage	Process Required
1	Identify the time-consuming program routines. You can make the greatest program performance improvement only if you multitask most of the work. Use the FLOWTRACE option or the SPY utility, and execute the resulting program on one CPU to determine which routines take the most execution time.
2	Form a static calling tree to better understand the call relationships among the subroutines. Use the TREE=FULL option with the FTREF utility (see section 7, Debugging) to obtain a static calling tree listing.

Stage	Process Required
3	Form a dynamic calling tree (using the output of FLOWTRACE or SPY) for those parts of the static calling tree containing the time-consuming subroutines. A dynamic calling tree shows the looping structures contained in and containing the time-consuming subroutines. CFT and CFT77 produce tables describing statement labels that can aid in the location of DO and IF loops.
4	Identify loops with sufficient work granularity to be considered for multitasking. The FLOWTRACE or SPY output is an aid to estimating the work size for loops containing calls to subroutines, and it may also help in estimating work granularity of loops within subroutines if you know the loop bounds. The minimum granularity that may be accepted depends on whether macrotasking or microtasking is used. (Microtasking is usually the better choice with small granularity.)
5	Eliminate loops with obvious data or control dependencies that prohibit them from being multitasked. The outer time-step loop is an example in which results computed on one iteration are inputs for the next iteration.
6	Choose the outermost loop of each nested set of loops that you are now considering for multitasking, and proceed to subsection 6.3.2, Verifying and Creating Independence. (The nested set of loops is one you have identified for further analysis; it may include calls to subroutines containing lower-level loop nests.)

6.3.2 VERIFYING AND CREATING INDEPENDENCE

The goal of the second step is to understand the use of variables referenced in each of the nested sets of loops identified in step 1. The purpose is to verify that computational, storage, and temporal independence are present or that the program can be modified to create the independence required for multitasking.

The stages in understanding the use of variables are as follows:

Stage	Process Required
1	With the multitasking model in mind, identify the new scope boundary for the loop being considered for multitasking.

Stage	Process Required
2	Record all variables referenced within the new scope boundary. Use the CB=FULL option with the FTREF utility to analyze the use of COMMON block variables.
3	Determine the computational independence of each variable by assessing its data independence for all control paths through the iterations. Ask the questions, "Is the variable produced by another iteration?" and, "Is the variable used by another iteration?" The answers to both must be "no," either now or after stage 5.
4	Evaluate each variable for storage independence according to its multitasked use. Will the variable be shared (one copy) or private (one copy for each task)? Compare the scope of each variable with the new scope boundary introduced by multitasking. Take care to follow the spread of scope through COMMON and argument lists, and watch out for equivalences.
5	Modify the old code with the new scope required for multitasking. Place shared variables in COMMON blocks if they are shared among tasks or in TASK COMMON blocks if they are local to a task. Remove private variables from COMMON blocks, being attentive to the ramifications these changes have on other parts of the code. Do the changes made for storage independence now create the computational independence required in stage 3? If not, return to stage 3 and repeat; otherwise, continue to stage 6.
6	Maintain determinism and temporal independence, recognizing the possible need to synchronize the tasks at the start and end of inner loops. Also note which variables require monitoring.
7	Perform the preceding stages for each set of nested loops. If you find a dependence that cannot be removed, return to stage 1 in subsection 6.3.1, Locating Potential Parallelism, choose the DO loop at the next-lower level, and repeat the stages of this subsection. If independence is guaranteed, proceed to subsection 6.3.3, Writing Multitasked Code.

6.3.3 WRITING MULTITASKED CODE

You have made most of the decisions concerning writing the multitasked code during the previous analysis steps, reorganizing the storage of variables according to their multitasked use. You can use the DOALL model (described in section 9, Advanced Macrotasking in Fortran) on each set of nested loops to form a task, or you can combine several sets into one larger task. Choose either static or dynamic partitioning of iterations and insert mechanisms for synchronization where necessary. Choose locks for, or guard, each critical region of code to be protected.

The macrotasking examples presented in this manual and the previously outlined procedure employed a particular programming style. Work was statically partitioned into *p* components when converting programs for multitasking on *p* processors. Each component was intended to be executed simultaneously by a separate processor, and synchronization followed each partitioned segment of work.

The microtasking examples presented employ a dynamic partitioning scheme. Work is organized so that tasks execute the next piece of work to be done and there is no reliance on any particular number of processors.

These associations (macrotasking/static and microtasking/dynamic) seem to be natural ones, but it is also possible to do dynamic partitioning with macrotasking or static partitioning with microtasking. You should use the partitioning technique and feature most appropriate to your application and with which you are most comfortable.

A final inspection of the code should check to see that the multitasking mechanisms employed have been properly initialized, that event, lock, or barrier names or guard numbers are shared among the tasks that use them, and that all tasks will have completed at the job's end. Verify that the program's results will not be affected by actually having fewer tasks available than you might want.

6.4 MULTITASKING I/O

The major concern throughout this manual is to multitask computations. For some applications, attention to I/O is at least as important as the speedup of computation.

It is possible to multitask I/O on a limited basis. Different tasks can perform I/O on different files. I/O on the same file by different tasks is limited by the nondeterministic nature of task execution. Parts of the I/O support library are critical regions that are protected from simultaneous access and thus limit the parallelism that you can exploit.

6.5 MINIMIZING MEMORY CONTENTION

One factor influencing performance in a shared memory architecture is memory contention. Memory references made with the three computational ports of each processor may result in resource conflicts. These conflicts are resolved by having some references wait until the required resource becomes available. This waiting can cause a computation on a single CPU to take longer in a system with all processors busy than it would in an otherwise quiet environment.

The user has control over the intensity of memory references as reflected in the programming style. The intensity of vector memory references can be measured in units of memory references per floating-point operation (memrefs/flop). The layout of the memory references across the interleaved memory banks is also important.

Programming styles that reduce the memory contention increase performance both in uniprocessing and multitasking environments. The following is a list of program optimizations that reduce memory contention. As with any optimizations, the following are of value only until the compiler can do better in CAL than the user can in Fortran.

1. VERTICAL INNER-LOOP UNROLLING

```
      DO 10 J = 1, M
        DO 10 I = 1, 2*N
  10      A(I,J) = A(I,J) + B(I,J)

      DO 11 J = 1, M                    MAXIMIZE CHIME OVERLAP
C                                       NO REGISTER RESERVATION

        DO 11 I = 1, 2*N, 2
          A(I  ,J) = A(I  ,J) + B(I  ,J)
  11      A(I+1,J) = A(I+1,J) + B(I+1,J)
```

2. HORIZONTAL INNER-LOOP UNROLLING

```
      DO 20 J = 1, M                    3 MEMREFS / 1 FLOP
        DO 20 I = 1, 3
  20      A(J) = A(J) + B(I,J)

      DO 21 J = 1, M                    5 MEMREFS / 3 FLOPS
  21    A(J) = A(J) + B(1,J) + B(2,J) + B(3,J)
```

3. VERTICAL OUTER-LOOP UNROLLING

```
      DO 30 J = 1, 3                    4 MEMREFS / 2 FLOPS
        DO 30 I = 1, N
30        A(I,J) = (A(I,J) + B(I,J)) / C(I)

        DO 31 I = 1, N                 10 MEMREFS / 6 FLOPS
          A(I,1) = (A(I,1) + B(I,1)) / C(I)
          A(I,2) = (A(I,2) + B(I,2)) / C(I)
31        A(I,3) = (A(I,3) + B(I,3)) / C(I)
```

4. HORIZONTAL OUTER-LOOP UNROLLING

```
      DO 40 J = 1, 4*M+1                3 MEMREFS / 2 FLOPS
        DO 40 I = 1, N
40        Y(I) = Y(I) + X(J) * M(I,J)

      DO 41 J = 1, 4*M+1, 4             6 MEMREFS / 8 FLOPS
        DO 41 I = 1, N
41        Y(I) = Y(I) + X(J  ) * M(I,J  )
     $                + X(J+1) * M(I,J+1)
     $                + X(J+2) * M(I,J+2)
     $                + X(J+3) * M(I,J+3)
```

5. DIMENSIONING OF ARRAYS

```
      DIMENSION  X(64,100)
      DO 50 I = 1, 64               X(I,*) IN  1 BANK
        DO 50 J = 1, 100
50        X(I,J) = X(I,J) + 1.0

      DIMENSION  X(65,100)
      DO 51 I = 1, 64               X(I,*) IN 64 BANKS
        DO 51 J = 1, 100
51        X(I,J) = X(I,J) + 1.0
```

6. LOOP INTERCHANGE

```
      DIMENSION  X(64,64)
      DO 60 I = 1, 64               X(I,*) IN  1 BANK
        DO 60 J = 1, 64
60        X(I,J) = X(I,J) + 1.0

      DIMENSION  X(64,64)
      DO 61 J = 1, 64               X(*,J) IN 64 BANKS
        DO 61 I = 1, 64
61        X(I,J) = X(I,J) + 1.0
```

7. PADDING BETWEEN COMMON BLOCK ARRAYS

```
      COMMON / Z /  X(2,1000), Y(2,1000)
      DO 70 I = 1, 2              X(I,*) IN BANKS 0,2,4,6,8,...
                                  Y(I,*) IN BANKS 0,2,4,6,8,...
        DO 70 J = 1, 1000
 70       X(I,J) = Y(I,J) + 1.0

      COMMON / Z /  X(2,1000), DUMMY, Y(2,1000)
      DO 71 I = 1, 2              X(I,*) IN BANKS 0,2,4,6,8,...
                                  Y(I,*) IN BANKS 1,3,5,7,9,...
        DO 71 J = 1, 1000
 71       X(I,J) = Y(I,J) + 1.0
```

7. DEBUGGING

This section provides information on techniques and tools that can help you to debug your multitasked program.

Like any other computer program, a multitasked application may fail for any of a number of reasons. Multitasking introduces two further complexities:

- A new set of potential problem areas in timings and synchronization
- Increased difficulty in analyzing and resolving any type of problem

The following list includes a few of the reasons a multitasked program could fail:

- Deadlock between tasks
- Errors in arithmetic, coding, or algorithm unrelated to multitasking
- Failure to protect critical regions
- Failure to provide for local variables
- Failure to consider the differences between stack and static local variables
- Lack of synchronization between tasks
- Violations of data dependence relationships

7.1 FREQUENT ERRORS

Use the following checklist to help avoid common errors when converting code to be multitasked. For microtasking, also review the rules described in subsection 4.5, Rules to Follow.

1. Properly declare and initialize macrotasking mechanisms by doing the following:
    - Set the task identifier array to a size of 2 or 3, and declare it as an integer type.
    - Initialize the first element of the task identifier array.

- Declare subroutine tasks to be EXTERNAL.

- Ensure that the argument list passed to TSKSTART does not contain expressions or local variables whose locations can be reused before they are accessed by the new task.

- Ensure that all task control arrays are accessible to the tasks that use them.

- Ensure that all event, lock, and barrier variables have been assigned before use and are accessible to tasks that use them.

- Ensure that any necessary initialization of events is done.

- Ensure that any necessary initialization of locks is done.

2. Use macrotasking mechanisms correctly, checking the following items:

   - Ensure that every TSKSTART has a corresponding TSKWAIT.

   - Ensure that every EVPOST has a corresponding EVWAIT and EVCLEAR.

   - Ensure that every LOCKON has a corresponding LOCKOFF.

   - Ensure that all tasks do appropriate BARSYNC calls.

3. Guarantee the independence of tasks and their data by doing the following:

   - Ensure that tasks do not rely on quantities computed in other tasks, except between synchronization points.

   - Ensure that tasks compute variables stored in separate storage locations, except for monitored variables.

   - Use sychronization points when the order of execution of tasks is important; otherwise, ensure that the execution order of tasks is immaterial.

4. Handle Fortran local variables properly.

   - Ensure that subroutines do not expect the value of a local variable to be preserved between calls.

   - Ensure that subroutines do not expect a local variable to begin with a particular value (especially zero).

- Ensure that local variables are not used as arguments in TSKSTART calls, because the lifetime of the task being started might exceed the lifetime of the local variable.

- Ensure that expressions are not used as arguments in TSKSTART calls, because their storage may be reused by the compiled code before the task being started can access the values in the expression.

- Heed warnings about static allocation of variables in DATA statements, and make code changes if the variables need to be allocated on the stack.

5. Ensure that all of the software is reentrant or locked.

   - Specify ALLOC=STACK in the CFT or CFT77 control statement.

   - Use the multitasked version of the default system text ($SYSTXT in COS, **asdef** in UNICOS) when CAL code is assembled.

   - Access the multitasked versions of all necessary run-time libraries at load time. The COS libraries are $ARLIB, $FTLIB, $IOLIB, $SCILIB, $SYSLIB, and $UTLIB. The UNICOS libraries are libm, libf, libio, libsci, and libu.

7.2 PERFORMANCE ERRORS

The performance speedup of a multitasked program may sometimes be less than expected. Several causes are possible, including the following:

1. Only a small fraction of the single-processor execution time was multitasked. Although the program was running on a system with multiple processors, the program did not make significant use of these resources. Multitasking is an optimization that produces performance speedup only to the code to which it is applied. See subsection 2.12, Amdahl's Law and Theoretical Speedup. ACTION: Convert more of the program to be multitasked.

2. The multitasking overhead was too large for the granularity of parallelism exploited. The added code for multitasking and synchronization delays slowed the program down more than the parallel execution speeded it up.
   ACTION: Increase the granularity by multitasking outermost loops, use microtasking for this code, or use multitasked CAL code to decrease the overhead.

3. Multitasking was employed at the expense of vectorization. Partitioning an inner vectorizable loop into tasks for multitasking may shorten the original vector length. The speedup from multitasking was less than the slowdown caused by reduced vector length.
   ACTION: Multitask outer loops and vectorize inner loops, or see the variants for microtasking's DO GLOBAL directive (described in subsection 4.3.7).

4. The workload for all processors was not balanced. The distribution of work among the processors was not shared equally, causing some processors to wait unproductively for others to finish.
   ACTION: Use dynamic work balancing (see section 3, Multitasking Basics).

5. The library routine TSKTUNE was used incorrectly. Setting 'MAXCPU' to a value less than the number of physical processors limits the resources available to the program. Setting 'DBRELEAS' to a value less than the number of physical processors minus 1 may cause excessive overhead in allocating and deallocating processors to the program. A small value of 'HOLDTIME' has a similar effect. A large value of 'SAMPLE' prevents the the multitasking library from taking immediate action when a task becomes ready.
   ACTION: Choose the appropriate values for the TSKTUNE arguments, or use the default values.

7.3 COS TASKS VERSUS USER TASKS

When looking at job output from a multitasked job, do not be confused by COS uses of the term *task*. Appendix F, Design Description, introduces the concept of the logical CPU. The references in COS-related output to *task* refer to a logical CPU and not to the task or tasks created by the user. The number of COS tasks may be equal to the number of user tasks, but the user tasks may have been assigned over time to one, several, or all of the COS tasks.

Output such as the following may lead to this confusion:

- CHARGES. If multiple COS tasks are created during the lifetime of a user job, CHARGES generates time executing in CPU, time waiting for CPU, and time waiting for I/O values for each task and for the total job.

- DUMP. If multiple COS tasks are active when a DUMPJOB is executed, the DUMP output shows an Exchange Package for each task, starting with the task in execution when the DUMPJOB was executed.

- DEBUG. If multiple COS tasks are active when a DUMPJOB executes, DEBUG traces back through each library task that was executing, starting with the task in execution when the DUMPJOB executed. Library tasks not assigned to COS tasks at the time DUMPJOB executes are not reported by default, but they are reported when the TASKS option is used (see the subsection on DEBUG later in this section).

7.4 CONDITIONAL MULTITASKING

Section 6, Program Analysis and Conversion, describes a technique that allows a multitasked program to be easily run in a nonmultitasked mode. You can isolate some types of multitasking errors this way. With microtasking, specify 1 on the CMIC$ GETCPUS directive to run in nonmultitasked mode.

7.5 ELIMINATING OPERATING SYSTEM MULTITASKING

A multitasked program can be run in a single-tasked mode. You accomplish this by using the multitasking tuning call TSKTUNE (see appendix F, Design Description) to set the maximum number of logical CPUs to one. The library scheduler continues to execute and processes multiple tasks, but only one task is known to the operating system.

In effect, this leads to tasks being executed largely in sequential order, with task switches only occurring when the executing task becomes blocked. The sequence of task execution is reproducible from run to run. This procedure can help with the following types of problems:

- Problems not related to multitasking. The problem should still occur but with less complexity surrounding the issue.

- Synchronization problems caused when two tasks simultaneously enter critical regions. Eliminating COS multitasking partially remedies this problem.

7.6 FTREF - FORTRAN CROSS-REFERENCE

FTREF is a general Fortran analysis tool described in the COS Performance Utilities Reference Manual and in the UNICOS Performance Utilities Reference Manual. It processes the listing from a CFT77 or CFT program and produces the following outputs:

- For each COMMON block, the name of each module that uses it.

- For each COMMON block, a detailed cross-reference, by variable, of the module and the lines that reference it.

- For each module, information on its entry points, which modules it calls, which modules call it, and COMMON blocks it uses.

- A static calling tree, displaying module calls in a graphic manner through the use of indentation. The tree can be started at any point, allowing for the display of specific subtrees. The tree can be stopped at any routine; also, a tree or subtree of a specific depth can be requested.

- Information on uses of the major macrotasking routines. For locks, events, and barriers,† these are ordered by the name of the lock or event variable. For task calls, these are presented in source order.

- Information about whether COMMON variables are locked when they are referenced or redefined.

- Information about whether nonreentrant subroutine calls are locked.

FTREF is of particular value for multitasking analysis because it can collect and consolidate information on global variables and their use by the subroutines within a Fortran application.

The CHKBLK and CHKMOD directives are also useful during the conversion process. CHKBLK detects whether COMMON variables are protected when referenced or redefined. CHKMOD detects whether calls to nonreentrant routines are locked.

The following example shows some of the listing options specific to multitasking. The first three listings result from the MULTI control statement parameter and show calls to the major multitasking routines. The fourth listing is output by the CHKBLK directive, which has referenced BLK3, BLK4, and BLK5. The listing indicates which references appear to be locked and which are unlocked. Some potential error conditions are flagged by special characters, as noted in the following example. The final listing is output by the CHKMOD directive, which references a routine called NONREENT. FTREF lists all calls to the routine and also includes information about whether the calls are locked.

† Barrier support in FTREF is deferred.

```
                                    ************************************
                                    *                                  *
                                    *   SUMMARY OF TASK ROUTINE USAGE  *
                                    *                                  *
                                    ************************************

        TSKSTART                              TSKWAIT                 TSKTUNE
MODULE     STMT.#   EXTERNAL      MODULE     STMT.#       MODULE     STMT.#

MAINPRG         8       SUB1      MAINPRG        29       MAINPRG        14
MAINPRG         9       SUB1
MAINPRG        10       SUB1


                                    ******************************
                                    *                            *
                                    *   SUMMARY OF LOCK USAGE    *
                                    *                            *
                                    ******************************

  LOCK          LOCKASGN            LOCKON              LOCKOFF             LOCKREL
VARIABLE     MODULE   STMT.#    MODULE    STMT.#    MODULE    STMT.#    MODULE    STMT.#    BLOCKS LOCKED

LK1          MAINPRG      13    MAINPRG       17    MAINPRG       22    none              BLK3     BLK4     BLK5
                                MAINPRG       31    SUBROUT       14
                                SUBROUT       10

  LK2            none          SUBROUT       16    SUBROUT       20    none              BLK4


                                    ******************************
                                    *                            *
                                    *   SUMMARY OF EVENT USAGE   *
                                    *                            *
                                    ******************************

 EVENT           EVASGN              EVPOST              EVWAIT              EVCLEAR             EVREL
VARIABLE     MODULE   STMT.#    MODULE    STMT.#    MODULE    STMT.#    MODULE    STMT.#    MODULE    STMT.#

EV1          MAINPRG      11    MAINPRG       15    MAINPRG       18    MAINPRG       16    MAINPRG       30
                                MAINPRG       23    MAINPRG       25    MAINPRG       24
                                SUBROUT        6    SUBROUT       13    SUBROUT        8

EV2          MAINPRG      12    SUBROUT       15    SUBROUT       22      none                none

EV3          ASUB2         3    SUBROUT       23    SUBROUT       25    SUBROUT       24    none
                                                   ASUB2          5    ASUB2          4

EV4            none              none              SUBROUT       19      none                none


                                    ****************************************
                                    *                                      *
                                    *   LOCKED VARIABLES CROSS REFERENCE   *
                                    *                                      *
                                    ****************************************

BLOCK NAME

 BLK3

  ADDRESS NAME     MODULE                SOURCE PROGRAM REFERENCE

        0 DATA4A   MAINPRG      LOCKED :     32$
                              UNLOCKED :     26*
        0 DATA4A   SUBROUT      LOCKED :     17
                              UNLOCKED :     21*

        1 DATA4B   MAINPRG      LOCKED :     32$     19
                              UNLOCKED :     26

 BLK4

  ADDRESS NAME     MODULE                SOURCE PROGRAM REFERENCE

        1 DATA5B   MAINPRG      LOCKED :     20

 BLK5

  ADDRESS NAME     MODULE                SOURCE PROGRAM REFERENCE

       **** No references ****

        Note : *  store but not locked
               $  LOCKON never LOCKOFF
               P  used as a parameter


                                    ****************************************
                                    *                                      *
                                    *   LOCKED EXTERNALS CROSS REFERENCE   *
                                    *                                      *
                                    ****************************************

NONREENT   CALLED BY            SOURCE PROGRAM REFERENCE

           MAINPRG      LOCKED :     21
           SUBROUT      LOCKED :     11
                      UNLOCKED :      7

      Note : *  store but not locked
             $  LOCKON never LOCKOFF
             P  used as a parameter                                    1667
```

7.7 FLOWTRACE

The FLOWTRACE package, described in the COS Performance Utilities Reference Manual is useful when analyzing programs for multitasking (see section 6, Program Analysis and Conversion).

COS 1.17 is the first operating system to support multitasking flowtrace, in which times are accumulated for each subprogram within each task. If a subprogram MSUB is called by *n* distinct tasks, it has *n* distinct entries in the final report. At present, UNICOS does not support multitasking flowtrace.

7.8 SPY†

The SPY utility provides functionality similar to that of FLOWTRACE. On a nonmultitasked program, SPY provides information on heavily used subroutines (including library routines) and on code sections within subroutines.

SPY can also be used with a multitasked program. The usage results apply to the program as a whole and not on a task-by-task basis. For example, if one routine accounts for 50% of the program's execution time before multitasking, SPY shows 50% for the same routine in the multitasked version.

7.9 INTERPRETING TRACEBACKS

If a multitasked job aborts, the traceback reflects the last entry into the multitasking package. The following example demonstrates such a traceback:

```
AR004 - BAD SCALAR ARGUMENT TO ARLIB MATH ROUTINE
TB001 - BEGINNING OF TRACEBACK
      - $TRBK    WAS CALLED BY ARERP%    AT   24771b
      - ARERP%   WAS CALLED BY SQRT%     AT   25425d
      - SQRT%    WAS CALLED BY MTASK2    AT     455d
      - MTASK2   WAS CALLED BY TSKSTART  AT   24215d
      - TSKSTART WAS CALLED BY MTASK1    AT     313b
TB002 - END OF TRACEBACK
```

If a stack overflow occurs on entry to a routine, $STKUFEX appears in the traceback before the call of the routine.

† Deferred UNICOS implementation

7.10 DEADLOCK DETECTION

The library scheduler detects software deadlock. If one or more tasks exist but all are queued for events, locks, or tasks, the library scheduler recognizes the condition and aborts. The library scheduler writes an error message to the log file and also writes a message and report of the status of each existing task to the standard output file. The library scheduler does not recognize whether a subset of all tasks is deadlocked, so the messages may not be generated until long after the deadlock situation arises.

7.11 DEBUG

DEBUG is a symbolic dump utility described in the Symbolic Debugging Package Reference Manual. DEBUG traces back through all tasks that were executing when the dump dataset was written or, if the TASKS option is used, through all existing tasks. A summary of the status of each existing task is written at the beginning of the DEBUG output.

DEBUG also reports heap and stack statistics† that can be used to tune initial sizes and increments of the heap and stack. These statistics are only useful, however, when the sizes used are smaller than the optimum values, because they reflect the actual sizes of the heap and stack rather the amount of much memory needed.

7.11.1 DEBUG INVOCATION STATEMENT

```
JOB,JN=DBG,T=15.
ACCOUNT,AC=       ,US=    ,UPW=    .
multi.
cft,debug,alloc=stack.
segldr.
copyd,i=$debug,o=syms.
rewind,dn=syms.
$abd.
EXIT.
DUMPJOB.
DEBUG,blocks,s=syms.
/EOF
```

† Deferred UNICOS implementation

7.11.2 MULTITASKED FORTRAN PROGRAM

```
      program test
      common /blk/ itsk(3,4),array(16),imem(4),ilock
      data array/1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1/
      external prt
      call tsktune('MAXCPU',4)
      call lockasgn(ilock)
      do 10 i = 1,4
      itsk(1,i) = 3
      itsk(3,i) = i
      imem(i) = i
      call tskstart(itsk(1,i),prt,imem(i))
      call lockon(ilock)
10    continue
      call tskwait(itsk(1,1))
      call tskwait(itsk(1,2))
      call tskwait(itsk(1,3))
      call tskwait(itsk(1,4))
      isum = imem(1)+imem(2)+imem(3)+imem(4)
      print *, isum
      end
      subroutine prt(id)
      common /blk/ itsk(3,4),array(16),imem(4),ilock
      isum = 0
      if (id.eq.2) call abort
      do 12 i = 1,4
      isum = isum + array((id-1)*4+i)
12    continue
      imem(id) = isum
      return
      end
```

7.11.3 DEBUG LISTING

```
 PAGE       1                                DEBUG 2.2
******* START OF SYMBOLIC DUMP *******


STATUS OF EXISTING TASKS

Internal                                                              # Tasks
task id   User defined task value  Task status                        waiting
-------   ----------------------   ----------------------            -------
      3   00000000000000000002     running on logical cpu      2          0

      1   00000000000000000000     waiting for lock  00000053502          0
                                   ILOCK    in  BLK
```

TASK STATISTICS

Existing tasks	2		
Total tasks	3		
Most tasks at once	3		
Logical CPUs acquired	2		
Logical CPUs active	1		

STACK STATISTICS

Total stacks	3		
Stacks with overflow	0		
Avg. no of overflows	0.00		
Greatest stack size	2048 (10)	000000004000 (8)	words
Smallest stack size	2048 (10)	000000004000 (8)	words
Average stack size	2048 (10)	000000004000 (8)	words
Initial stack size	2048 (10)	000000004000 (8)	words

HEAP STATISTICS

Current Heap Length	6766 (10)	000000015156 (8)	words
Words allocated	4492 (10)	000000010614 (8)	
High water mark	6766 (10)	000000015156 (8)	words
Low water mark	2272 (10)	000000004340 (8)	words
Allocated blocks	2		
Total Allocations	3		
Heap Expansions	2		
Heap Contractions	0		
Initial heap size	2272 (10)	000000004340 (8)	words
Heap increment size	2048 (10)	000000004000 (8)	words

MULTITASKING HISTORY TRACE BUFFER

Time	Task	Action	Action-dependent data
588405		suspend logical cpu	1
488142		spin-wait logical cpu	1
487819	1	detach from logical cpu	1
487290	1	begin wait to set lock	ILOCK in BLK
487029	3	attach to logical cpu	2
486453	1	attach to logical cpu	1
485915	1	detach from logical cpu	2
485088	3	start task	00000000000000000002
481460	2	complete task	00000000000000000001
480550	2	attach to logical cpu	1
477865	1	set lock, no wait	ILOCK in BLK
477317	1	attach to logical cpu	2
476832		acquire logical cpu	2
9151		request logical cpu	
8891	1	detach from logical cpu	0

```
          Time  Task  Action                    Action-dependent data
          ----  ----  ------                    ---------------------
          8045    2   start task                00000000000000000001
          1042    1   assign lock               ILOCK    in BLK
             0    1   start task                00000000000000000000
```

Key to Trace Buffer Display :

TIME is the difference between the real time clock, and the program starting time.

TASK is the unique internal task number assigned by the TSKSTART routine.

If ACTION involves:	Then ACTION DEPENDENT DATA is:
start task	user-defined task value
complete task	user-defined task value
TSKWAIT	internal ID of awaited task
lock	lock address
event	event address
logical CPU	logical CPU number

STATUS OF JOB DATASETS

Dataset Name	Alias Names	Status	Last Operation	Dataset Type	
$DEBUG		Open for i/o - At EOF	Read	COS Blocked	Sequential
$ABD		Open for input - At EOR	Read	COS Blocked	Sequential
SYMS		Open for i/o	Read	COS Blocked	Sequential
$SCILIB		Open for i/o - At EOF	Read	COS Blocked	Sequential
$PSCLIB		Open for i/o - At EOF	Read	COS Blocked	Sequential
$IN		Open for i/o - At EOF	Read	COS Blocked	Sequential
	FT05				
$OUT		Open for i/o - At EOF	Write	COS Blocked	Sequential
	FT06				

DISPLAYING TASK 3

The Symbolic Debug Routine was entered when your program stopped in routine ABORT with P address 000000011170a

No symbol table was found for this routine

Routine ABORT was called by PRT at address 000000000350a,line number 4

```
Displaying variables for Routine  PRT

D  ID                          2                          2b
   ISUM                        0                          0b
   I                           0                          0b


Displaying variables for Block    BLK

   ILOCK                      -9223275417270462496    1000002576000000053740

   ITSK                     Array ( 3,4 ) of Integer
   Value :  i =      1 to        3  (i,     1,     0,     0,   0,     0,     0)
   i=     1                     3                        2                    1
   Value :  i =      1 to        3  (i,     2,     0,     0,   0,     0,     0)
   i=     1                     3                        3                    2
   Value :  i =      1 to        3  (i,     3,     0,     0,   0,     0,     0)
   i=     1                     0                        0                    0
   Value :  i =      1 to        3  (i,     4,     0,     0,   0,     0,     0)
   i=     1                     0                        0                    0

   ARRAY                    Array ( 16 ) of Real
   Value :  i =      1 to       16  (i,     0,     0,     0,     0,     0,   0)
   i=     1       1.00000000000          1.00000000000          1.00000000000
   i=     4       1.00000000000          1.00000000000          1.00000000000
   i=     7       1.00000000000          1.00000000000          1.00000000000
   i=    10       1.00000000000          1.00000000000          1.00000000000
   i=    13       1.00000000000          1.00000000000          1.00000000000
   i=    16       1.00000000000

IMEM                        Array ( 4 ) of Integer
Value :  i =       1 to        4  (i,     0,     0,     0,     0,     0,     0)
i=     1                       4                        2                    0
i=     4                       0

Routine PRT was called by TSKSTART at address 000000031616c,line number 11

No symbol table was found for this routine


Routine TSKSTART was called by TEST at address 000000032047d,line number 11


Displaying variables for Routine  TEST

   I                          2                          2b
   ISUM                       0                          0b


***** END OF SYMBOLIC DUMP *****
```

7.12 LIBRARY DEBUG ROUTINES

Routines DUMP, PDUMP, and CRAYDUMP are all reentrant, but you should put locks around calls to these routines if they use the same datasets in different tasks so that the output from different tasks is not interspersed.

Routines TRBK, SNAP, and DUMPJOB are not locked and are not reentrant. If one of these routines is called from more than one task, locks should be placed around the calls.

8. MULTITASKING HISTORY TRACE BUFFER

The multitasking history trace buffer is a circular buffer in user memory that maintains a log of the actions taken by the multitasking library routines. Access to this buffer is through calls to library routines or through DEBUG after the program terminates. This buffer, each time it is full, may be dumped to a dataset specified in a call to subroutine BUFTUNE. The MTDUMP program can examine this unformatted dump of the buffer and display all of it in any of several possible formats, or it can search the dataset for specific entries. The buffer is large enough to record 1000 actions.

8.1 DEBUG DISPLAY

After a job completes execution, either normally or through an abort, DEBUG writes a formatted display of the contents of the multitasking history trace buffer left in memory. The MTBUF control statement parameter can specify the number of entries to display, beginning with the most recent and working backward. By default, as many entries as will fit on a single page of the DEBUG listing are displayed. Using MTBUF without a value lists all entries in the buffer. When MTBUF is used with a value, the number of entries printed will be equal to or less than the value, depending on the number of entries in the buffer.

Example of the Job Control Language (JCL) that uses DEBUG:

```
*
MULTI.
CFT,L=0,ALLOC=STACK.
SEGLDR.
$ABD.
DUMPJOB.
*  Print last 15 entries from the multitasking history trace buffer.
DEBUG,TRACE=0,MTBUF=15.
EXIT.
DUMPJOB.
*  If the job aborts, then print all entries in the buffer.
DEBUG,TRACE=0,MTBUF.
```

8.2 USER-LEVEL ROUTINES

The user-level routines for the multitasking history trace buffer can be called from a user program to control what is recorded in the buffer and to dump the contents of the buffer to a dataset.

8.2.1 BUFTUNE: SELECT ACTIONS TO BE RECORDED

BUFTUNE modifies the parameters that control which multitasking actions are recorded in the multitasking history trace buffer. You can call it any number of times. If it is not called, or before it is called for the first time, default parameter values are used.

Before BUFTUNE is called, all actions involving tasks, locks, events, logical CPUs, and users are recorded except for actions involving the Fortran I/O lock, which are ignored. A call to BUFTUNE specifying the TASKS, LOCKS, EVENTS, CPUS, or USERS keyword affects only the actions associated with that keyword. The ACTIONS option overrides what has been requested through TASKS, LOCKS, EVENTS, CPUS, and USERS.

Call from Fortran:

```
CALL BUFTUNE(keyword,value[,string])
```

keyword ASCII string, left-justified and blank-filled

value Either an integer or an ASCII string (left-justified and blank-filled), depending on the keyword

string A 24-character string (left-justified and blank-filled) used only with the keyword INFO

The valid keywords and their associated functions and meanings are as follows:

Keyword	Meaning
DN	The value of the DN keyword is the dataset specified to receive a dump of the multitasking history trace buffer. DN itself directs this dump of the buffer to the dataset. If you call BUFTUNE without the DN parameter, the multitasking history trace buffer is not dumped to any dataset.

Keyword	Meaning
FLUSH	Integer specifying the minimum number of unused entries allowed in the multitasking history trace buffer. When the number of unused entries falls below this level, the buffer is automatically flushed; that is, it is written to the dataset specified with the DN option. If DN is specified, the default FLUSH value is 40.
ACTIONS	Value is a 128-element integer array with a flag for each action that can be recorded in the multitasking history trace buffer. If the array element corresponding to a particular action is nonzero, that action is recorded; if the array element is zero, the action is ignored. The array indices (action codes) corresponding to each action are as follows:

Action Code	Action
1	Start a task
2	Complete a task
3	Call to TSKWAIT, no wait
4	Begin wait for task
5	Run after waiting for task
6	Test task
7	Assign lock
8	Release lock
9	Set lock
10	Begin wait to set lock
11	Run after waiting for lock
12	Clear lock
13	Test lock
14	Assign event
15	Release event
16	Post event
17	Clear event
18	Call EVWAIT, no wait
19	Begin wait for event
20	Run after wait for event
21	Test event
22	Attach to logical CPU
23	Detach from logical CPU
24, 25	Request a logical CPU

Keyword	Meaning	
ACTIONS (continued)	Action Code	Action
	26	Acquire a logical CPU
	27, 28	Delete a logical CPU
	29, 30	Suspend a logical CPU
	31, 32	Activate a logical CPU
	33	Begin spin-wait for a logical CPU
	34	Assign barrier
	35	Release barrier
	36	Call BARSYNC, no wait
	37	Begin wait at barrier
	38	Run after wait for barrier†
	39 - 64	(Reserved for future use)
	65 - 128	(Reserved for user access; see BUFUSER subroutine.)

INFO The value for this parameter is the integer user action code (65 through 128).

string This is an extra parameter available with the INFO keyword. The string for this parameter is a 24-character information string, unique to each action, entered by the user and printed for each user action code that is dumped.

BUFUSER lets you add entries to the multitasking history trace buffer. When the multitasking history trace buffer is dumped using DEBUG, BUFPRINT, or MTDUMP, this 24-character information string is dumped along with each action. This information must be available early in the program so that the strings can be written to the dump dataset for processing by MTDUMP. The INFO keyword does not turn these actions on to be recorded. They are normally on by default, but if previously turned off by the user, they may be reactivated with the ACTIONS or USERS keyword in a BUFTUNE call.

TASKS If *value*='ON'H, the actions with numbers 1 through 6 are recorded; if *value*='OFF'H, those actions are ignored. The default is 'ON'H.

LOCKS If *value*='ON'H, the actions with numbers 7 through 13 are recorded; if *value*='OFF'H, those actions are ignored. The default is 'ON'H.

† Deferred implementation

Keyword	Meaning
EVENTS	If *value*='ON'H, the actions with numbers 14 through 21 are recorded; if *value*='OFF'H, those actions are ignored. The default is 'ON'H.
CPUS	If *value*='ON'H, the actions with numbers 22 through 33 are recorded; if *value*='OFF'H, those actions are ignored. The default is 'ON'H.
USERS	If *value*='ON'H, the actions with numbers 65 through 128 are recorded; if *value*='OFF'H, those actions are ignored. The default is 'ON'H.
FIOLK	If *value*='ON'H, actions affecting the Fortran I/O lock are recorded; if *value*='OFF'H, they are ignored. This lock is used by library routines that handle Fortran reads and writes. The default is 'OFF'H.

The following BUFTUNE examples dump actions 1 through 6 to the dataset DMPFILE, which is the same as the function that TASKS performs.

```
* Turn on task actions, turn everything else off
INTEGER ACTION(128)
DATA ACTION/6*1,122*0/
CALL BUFTUNE('DN'H,'DMPFILE'H)
CALL BUFTUNE('ACTIONS'H,ACTION)
```

or

```
* Turn on task actions, turn everything else off
CALL BUFTUNE('DN'H,'DMPFILE'H)
CALL BUFTUNE('TASKS'H,'ON'H)
CALL BUFTUNE('LOCKS'H,'OFF'H)
CALL BUFTUNE('EVENTS'H,'OFF'H)
CALL BUFTUNE('CPUS'H,'OFF'H)
```

8.2.2 BUFPRINT: FORMATTED DUMP OF TRACE

BUFPRINT writes a formatted dump of the contents of the multitasking history trace buffer to a specified dataset. Actions are reported in chronological order.

Call from Fortran:

```
CALL BUFPRINT(empty[,dn])
```

empty Set this integer flag to zero if you want the contents of the buffer to be left unchanged or nonzero if the buffer is to be emptied (zeroed out) after its contents are printed

dn Name of the dataset to which a formatted dump of the contents of the multitasking history trace buffer is to be written. If none is specified, $OUT is used.

The following example of BUFPRINT zeroes out the buffer after its contents are written to $OUT:

```
IEMPTY = 1
CALL BUFPRINT(IEMPTY)
```

8.2.3 BUFDUMP: UNFORMATTED DUMP OF TRACE

BUFDUMP writes an unformatted dump of the multitasking history trace buffer to a specified dataset. MTDUMP can later use this dataset to provide formatted output of the dataset's contents or to let you examine the dataset. BUFDUMP adds a special entry if the buffer has overflowed, losing some entries.

Call from Fortran:

```
CALL BUFDUMP(empty,dn)
```

empty Set this integer flag to zero if you want the contents of the buffer to be left unchanged or to nonzero if the buffer is to be emptied (zeroed out) after its contents are printed

dn Name of the dataset to which an unformatted dump of the contents of the multitasking history trace buffer is written. To use the default dataset passed to BUFTUNE, a zero must be entered, but if no dataset is specified through BUFTUNE, the BUFDUMP call is ignored.

8.2.4 BUFUSER: ADD USER ENTRIES TO TRACE

BUFUSER lets you add entries to the multitasking history trace buffer.

Call from Fortran:

```
CALL BUFUSER(action,data)
```

action A numerical code that determines the action to be recorded in the buffer. The action codes 65 through 128 are reserved for this. The codes and their associated actions follow:

1 - 64	You cannot add entries with these action codes; the program is aborted.
65 - 128	This action code is compared to the action codes specified in BUFTUNE, either explicitly by the user or turned on by default, and if the action code appears in the BUFTUNE call, or is on by default, the corresponding entry is added to the multitasking history trace buffer. If the action code does not appear in the BUFTUNE call, this action/entry is ignored. If a string is provided (see BUFTUNE), it is dumped into the action field of the output for this entry. If no string is provided, the (decimal) action code is dumped into the action field. In either case, the *data* is written in octal (and ASCII, if it is a legal character) to the action-dependent data field of the output.
>128	The action code and *data* are added unconditionally (regardless of any BUFTUNE parameters) to the action code and action-dependent data fields of the output. They are both dumped in octal form (and in ASCII if it is a legal character).

data The only restriction on this value is that it be a single machine word. If it is a string, it must be left-justified, blank-filled, and less than 9 characters in length. It is dumped to the output in octal (and ASCII, if it is a legal character).

8.2.5 MTDUMP: EXAMINE TRACE DATASET

MTDUMP is a tool that examines an unformatted dump of the multitasking history trace buffer, which has been written to a dataset specified by a call to BUFTUNE with the DN keyword. MTDUMP lets you display the multitasking events in any of several useful formats.

The multitasking history trace buffer is dumped by calling BUFDUMP at any point in execution, or, if BUFDUMP is never called, the dump occurs automatically upon completion of the program. This automatic dump occurs only if BUFTUNE has been previously called with the DN keyword specifying a dataset to which to dump. If a dataset has not been specified, no dump is made and the only means of accessing the multitasking history trace buffer is through the DEBUG routine.

If BUFDUMP is not called during execution and the program aborts rather than running to completion, the dump of the multitasking history trace buffer does not occur and DEBUG has to be used to dump it (see subsection 8.1, DEBUG Display).

8.2.5.1 COS format

The MTDUMP control statement under COS is as follows. In the following format, braces indicate optional parameters that may have one or more occurrences.

Format:

```
MTDUMP,DN=dn,L=ldn,FORMAT=f{:f},INTERVAL=int,TASK=t{:t},
  DATA=d{:d},ACTION=a{:a},TASKS,LOCKS,EVENTS,CPUS,USERS.
```

DN=*dn* Dataset name; name of the dataset containing the unformatted dump of the multitasking history trace buffer. This parameter is required.

L=*ldn* Listing dataset name; name of the dataset to which the output listing is to be written. Default is $OUT.

FORMAT=*f*{:*f*}
One to five formats in which the multitasking history trace buffer is to be displayed. The options are as follows:

TOTALS Summary of data found in multitasking history trace buffer

CHRON — Chronological display

SYNC — Display of synchronization points, with a separate column for each of up to 16 user tasks

CPU — Display of logical CPU use, with a separate column for each of up to 16 logical CPUs

STATUS — Display of the status of up to 16 user tasks in uniform time intervals

If MTDUMP is specified with no formatting options, TOTALS is the default.

INTERVAL=*int*
: The number of clock periods in each time interval in the STATUS format display; the default is 1,000,000. This parameter has no effect on other format displays.

The following parameters let you list selected groups of buffer entries. The default is that all actions relevant to a given display format are reported or accounted for.

TASK=*t*{:*t*}
: List of internal task identifier numbers for which buffer entries should be listed, with a maximum of 10 task identifiers allowed. The default is all tasks. Use this parameter only with CHRON formats.

DATA=*d*{:*d*}
: List of action-dependent data values to be searched for, with a maximum of 10 data values allowed. The default is to list entries for all data values. Use this parameter only with CHRON formats.

ACTION=*a*{:*a*}
: List of action codes of buffer entries to be listed, with a maximum of 40 actions allowed. The default is to list entries for all action codes unless one or more of the ACTION, TASKS, LOCKS, EVENTS, CPUS, and USERS parameters have been used. See subsection 8.2.1, BUFTUNE: Select Actions to be Recorded, for the action codes. Use this parameter only with CHRON formats.

TASKS — Actions involving tasks are listed; these include task starts, completions, waits, and tests. Use this parameter only with CHRON formats.

LOCKS Actions involving locks are listed. Use this parameter only with CHRON formats.

EVENTS Actions involving events are listed. Use this parameter only with CHRON formats.

CPUS Actions involving logical CPUs are listed. Use this parameter only with CHRON formats.

USERS Actions involving user codes (65 through 128) are listed. Use this parameter only with CHRON and SYNC formats.

8.2.5.2 UNICOS format

The UNICOS **mtdump** command line is in the following format. You can specify the options in any order, and the blank spaces between the options (such as **-f**) and the arguments are optional. The only required operand is *file*. All output goes to **stdout** (standard output).

```
mtdump [-f form] [-i intr] [-t tsks] [-d data]
       [-a act] [-T] [-L] [-E] [-C] [-U] file
```

-f *form* One to five formats, separated by commas (with no blank spaces) and listed in any order, in which the multitasking history trace buffer is to be displayed. The format options are as follows:

totals	Summary of data found in multitasking history trace buffer
chron	Chronological display
sync	Display of synchronization points, with a separate column for each of up to 16 user tasks
cpu	Display of logical CPU use, with a separate column for each of up to 16 logical CPUs
status	Display of the status of up to 16 user tasks in uniform time intervals

If **mtdump** is specified with no formatting options, totals is the default.

-i *intr* The number of clock periods in each time interval in the status format display; the default is 1,000,000. This option has no effect on other format displays.

The following options let you list selected groups of buffer entries. The default is that all actions relevant to a given display format are reported or accounted for.

-t *tsks* List of internal task identifier numbers, separated by commas, for which buffer entries should be listed, with a maximum of 10 task identifiers allowed. The default is all tasks. Use this parameter only with chron formats.

-d *data* List of action-dependent data values to be searched for, separated by commas, with a maximum of 10 data values allowed. The default is to list entries for all data values. Use this parameter only with chron formats.

-a *act* List of action codes of buffer entries to be listed, separated by commas, with a maximum of 40 actions allowed. The default is to list entries for all action codes unless one or more of the action, tasks, locks, events, cpus, and users parameters have been used. See subsection 8.2.1, BUFTUNE: Select Actions to be Recorded, for the action codes. Use this parameter only with chron formats.

-T If specified, actions involving tasks are listed; these include task starts, completions, waits, and tests. Use this parameter only with chron formats.

-L If specified, actions involving locks are listed. Use this parameter only with chron formats.

-E If specified, actions involving events are listed. Use this parameter only with chron formats.

-C If specified, actions involving logical CPUs are listed. Use this parameter only with chron formats.

-U If specified, actions involving user codes (65 through 128) are listed. Use this parameter only with chron and sync formats.

file Name of the file containing the unformatted dump of the multitasking history trace buffer. This parameter is required.

8.2.5.3 Tips on combining parameters for COS and UNICOS

The MTDUMP program currently provides five different display formats: CHRON, SYNC, TOTALS, CPU, and STATUS. It also provides eight action-selection parameters: TASK=, DATA=, ACTION=, TASKS, LOCKS, EVENTS, CPUS, and USERS.

The following points represent how these formats and action-selection parameters interact with each other:

- The five formats do not interact with the eight action-selection parameters. If you selected the CPU format and the TASKS parameter, for example, you would get two separate reports. The TASKS information would be represented in the CHRON format.

- If you choose multiple action-selection parameters, the information would be combined into one report in the CHRON format.

- The DATA= and TASK= parameters act as screens on top of the action selectors. Mixing these two is not recommended, and mixing them with the action-selection parameters requires caution. If you do mix them with action-selection parameters, EVENT(I) is output only if all of the following are true:

  - ACTION(I) is selected by any of the six action-selection parameters, or no action-selection parameter is chosen.

  - TAKS(I) is explicitly listed in the task list with the TASK parameter, or the TASK parameter is not chosen.

  - DATA(I) is listed with the DATA parameter or the DATA parameter, is not chosen.

8.3 EXAMPLES

This subsection contains examples of different output formats for the multitasking history trace buffer. A key to the terms used in each example follows the display. This example shows the different ways that MTDUMP can either dump the contents of the buffer dump dataset or allow selected groups of buffer entries to be listed.

The COS JCL for these examples (MTDUMP,DN=*dumpdn*,TASK=2:3:5.) is as follows:

```
MULTI.
CFT,ALLOC=STACK,L=0.
SEGLDR.
$ABD.
DUMPJOB.
*
MTDUMP,DN=DMPFILE,FORMAT=CHRON:SYNC:CPU:STATUS:TOTALS,INTERVAL=2000000.
MTDUMP,DN=DMPFILE,FORMAT=CHRON,EVENTS.
MTDUMP,DN=DMPFILE,FORMAT=CHRON,TASKS.
MTDUMP,DN=DMPFILE,FORMAT=CHRON,ACTION=1:2:4:5.
MTDUMP,DN=DMPFILE,FORMAT=CHRON,TASK=4:5:6.
MTDUMP,DN=DMPFILE,FORMAT=CHRON,DATA=47741:3.
*
/EOF

  PROGRAM TEST

*        Dump the buffer to a dataset when it is full.
  CALL BUFTUNE('DN'H,'DMPFILE'H)
  . . .
  STOP
  END
```

The UNICOS equivalent for the COS JCL above is as follows:

```
cft -a stack test.f
segldr -o test test.o segdir
test
od test
mtdump -f chron,sync,cpu,status,totals -i 2000000 DMPFILE
mtdump -E DMPFILE -f chron
mtdump -T -f chron
mtdump -a 1,2,3,4,5 -f chron
mtdump -t 4,5,6 -f chron
mtdump -d 47741,3 -f chron
```

8.3.1 FORMAT PARAMETER

The following COS JCL statement or UNICOS command line results in the outputs shown in the following seven pages of examples.

```
MTDUMP, DN=dumpdn, FORMAT=CHRON:SYNC:CPU:STATUS, INTERVAL=2000000.
mtdump -f chron,sync,cpu,status -i 2000000 dumpfile
```

Each example shows the output that results from one of the values of the FORMAT parameter.

8.3.1.1 Chronological display

FORMAT=CHRON in the COS JCL statement or -f chron in the UNICOS command line yields a chronological display of tasks and actions.

```
      Time  Task  Action                   Action-dependent data
      ----  ----  ------                   ---------------------
         0     1  start task               00000000000000000000
      1534     1  assign lock               00054413
      1944     1  assign event              00054414
      4760     2  start task               00000000000000000002
      5518     1  detach from logical CPU      0
      5771        request logical CPU
    756593        acquire logical CPU          2
    757094     1  attach to logical CPU        2
    757625     1  begin wait for event      00054414
    758162     1  detach from logical CPU      2
    758712     2  attach to logical CPU        2
    759218     2  post event                00054414
    759977     2  detach from logical CPU      2
    760539     2  attach to logical CPU        2
    761091     2  clear event               00054414
    762797     2  set lock, no wait         00054413
    763208     2  clear lock                00054413
    763565     2  complete task            00000000000000000002
    765130     1  attach to logical CPU        2
    765534     1  run after wait for event  00054414
    765964     1  post event                00054414
    766341     1  clear event               00054414
    768118     1  set lock, no wait         00054413
    768492     1  clear lock                00054413
    768978     1  TSKWAIT, no wait             2
    771918     3  start task               00000000000000000003
    772744     1  detach from logical CPU      2
    773268     1  attach to logical CPU        2
    776257     4  start task               00000000000000000004
    777055     1  detach from logical CPU      2
    777348        request logical CPU
   1180970     1  attach to logical CPU        1
   1184072     5  start task               00000000000000000005
   1184873     1  detach from logical CPU      1
   1185166        request logical CPU
   1932758     1  attach to logical CPU        1
   1935343        acquire logical CPU          3
   1935797     6  start task               00000000000000000006
   1935956     3  attach to logical CPU        3
   1936699     1  detach from logical CPU      1
   1937225     1  attach to logical CPU        1
   1937872     3  set lock, no wait         00054413
   1938257     3  clear lock                00054413
   1938609     3  complete task            00000000000000000003
```

Time	Task	Action	Action-dependent data
1940215	7	start task	00000000000000000007
1940973	1	detach from logical CPU	1
1941477	1	attach to logical CPU	3
1942046	4	attach to logical CPU	1
1943918	4	set lock, no wait	00054413
1944333	4	clear lock	00054413
1944539	8	start task	00000000000000000010
1944693	4	complete task	00000000000000000004
1945406	1	detach from logical CPU	3
1946282	1	attach to logical CPU	3
1946966	5	attach to logical CPU	1
1948847	5	set lock, no wait	00054413
1949270	5	clear lock	00054413
1949496	9	start task	00000000000000000011
1949652	5	complete task	00000000000000000005
1950431	1	detach from logical CPU	3
1951363	1	attach to logical CPU	3
1952095	6	attach to logical CPU	1
1953420	1	set lock, no wait	00054413
1953798	1	clear lock	00054413
1954088	6	set lock, no wait	00054413
1954354	1	begin wait for task	9
1954909	1	detach from logical CPU	3
1955100	6	clear lock	00054413
1955655	7	attach to logical CPU	3
1957531	7	set lock, no wait	00054413
1957908	7	clear lock	00054413
1958267	7	complete task	00000000000000000007
1959872	8	attach to logical CPU	3
1961732	8	set lock, no wait	00054413
1962100	8	clear lock	00054413
1962445	8	complete task	00000000000000000010
1964010	9	attach to logical CPU	3
1965905	9	set lock, no wait	00054413
1966276	9	clear lock	00054413
1966623	9	complete task	00000000000000000011
1968225	1	attach to logical CPU	3
1968589	1	run after wait for task	9
1969049	1	TSKWAIT, no wait	8
1969506	1	TSKWAIT, no wait	7
1969883	1	begin wait for task	6
2371705		acquire logical CPU	4
60212892	1	detach from logical CPU	3
60213188		spin-wait logical CPU	3
60313488		suspend logical CPU	3
60355394		spin-wait logical CPU	4
60455675		suspend logical CPU	4
60498138		spin-wait logical CPU	2
60598434		suspend logical CPU	2

```
      Time  Task  Action                    Action-dependent data
      ----  ----  ------                    ---------------------
  60645145     6  complete task             00000000000000000006
  60646811     1  attach to logical CPU          1
  60647184     1  run after wait for task        6
  60647607     1  TSKWAIT, no wait               5
  60647992     1  TSKWAIT, no wait               4
  60648397     1  TSKWAIT, no wait               3
  60684734     1  complete task             00000000000000000000
```

Key to information ----------

The time is from the real time clock, minus the starting time.

The task number given is the unique internal task identifier assigned by TSKSTART.

If action involves:	then action-dependent data is:
start task	user-defined task value
complete task	user-defined task value
TSKWAIT	internal task ID of waited-upon task
lock	lock address
event	event address
logical CPU	logical CPU number

8.3.1.2 Synchronization points

FORMAT=SYNC in the COS JCL statement or -f sync in the UNICOS command line yields a display of synchronization points, with a separate column for up to 16 user tasks.

```
          <-------- Internal Task Identifier ------->
    Time    1   2   3   4   5   6   7   8   9 ... 16
    ----   --  --  --  --  --  --  --  --  --     --
       0    $                                      00000000000000000000
    4760    *   $                                  00000000000000000002
  757094    *   !
  757625   e.   !                                  event at 00054414
  758712    .   *
  759218    !  P*                                  event at 00054414
  760539    !   *
  761091    !  C*                                  event at 00054414
  762797    !  (*                                  lock at  00054413
  763208    !  )*                                  lock at  00054413
  763565    !   $                                  00000000000000000002
```

```
          <-------- Internal Task Identifier ------->
   Time    1   2   3   4   5   6   7   8   9 ... 16
   ----   --  --  --  --  --  --  --  --  --     --
 765130    *
 765534    *                                        event at 00054414
 765964   P*                                        event at 00054414
 766341   C*                                        event at 00054414
 768118   (*                                        lock at  00054413
 768492   )*                                        lock at  00054413
 768978   t*                                        task    2
 771918    *       $                                00000000000000000003
 773268    *       !
 776257    *       !   $                            00000000000000000004
1180970    *       !   !
1184072    *       !   !   $                        00000000000000000005
1932758    *       !   !   !
1935797    *       !   !   !   $                    00000000000000000006
1935956    *       *   !   !   !
1937225    *       *   !   !   !
1937872    *      (*   !   !   !                    lock at  00054413
1938257    *      )*   !   !   !                    lock at  00054413
1938609    *       $   !   !   !                    00000000000000000003
1940215    *           !   !   !   $                00000000000000000007
1941477    *           !   !   !   !
1942046    *           *   !   !   !
1943918    *          (*   !   !   !                lock at  00054413
1944333    *          )*   !   !   !                lock at  00054413
1944539    *           *   !   !   !   $            00000000000000000010
1944693    *           $   !   !   !   !            00000000000000000004
1946282    *               !   !   !   !
1946966    *               *   !   !   !
1948847    *              (*   !   !   !            lock at  00054413
1949270    *              )*   !   !   !            lock at  00054413
1949496    *               *   !   !   !   $        00000000000000000011
1949652    *               $   !   !   !   !        00000000000000000005
1951363    *                   !   !   !   !
1952095    *                   *   !   !   !
1953420   (*                   *   !   !   !         lock at  00054413
1953798   )*                   *   !   !   !         lock at  00054413
1954088    *                  (*   !   !   !         lock at  00054413
1954354   t.                   *   !   !   !         task    9
1955100    .                  )*   !   !   !         lock at  00054413
1955655    .                   *   *   !   !
1957531    .                   *  (*   !   !         lock at  00054413
1957908    .                   *  )*   !   !         lock at  00054413
1958267    .                   *   $   !   !         00000000000000000007
1959872    .                   *       *   !
1961732    .                   *      (*   !         lock at  00054413
1962100    .                   *      )*   !         lock at  00054413
1962445    .                   *       $   !         00000000000000000010
1964010    .                   *           *
```

```
          <-------- Internal Task Identifier ------->
    Time    1   2   3   4   5   6   7   8   9 ... 16
    ----   --  --  --  --  --  --  --  --  --     --
 1965905    .                   *           (*         lock at  00054413
 1966276    .                   *           )*         lock at  00054413
 1966623    !                   *            $         00000000000000000011
 1968225    *                   *
 1968589    *                   *                      task    9
 1969049   t*                   *                      task    8
 1969506   t*                   *                      task    7
 1969883   t.                   *                      task    6
60645145    !                   $                      00000000000000000006
60646811    *
60647184    *                                          task    6
60647607   t*                                          task    5
60647992   t*                                          task    4
60648397   t*                                          task    3
60684734    $                                          00000000000000000000
```

```
Key to symbols used -------------------
 $  begin task/complete task
 *  running
(*  set lock
)*  clear lock
t*  TSKWAIT, no wait
e*  EVWAIT, no wait
P*  post event
C*  clear event
 !  ready; waiting for CPU
 .  waiting for lock, event, or task
l.  begin wait for lock
e.  begin wait for event
t.  begin wait for task
```

8.3.1.3 Logical CPU use

FORMAT=CPU in the COS JCL statement or -f cpu in the UNICOS command line outputs a table of logical CPU use, with a separate column for each of up to 16 logical CPUs.

```
        Logical CPU Number
  Time      1    2    3    4  . . . 16
  ----     --   --   --   --        --
    5518    !
  756593    !    !
  757094    !    1
  758162    !    !
  758712    !    2
  759977    !    !
```

```
        Logical CPU Number
  Time      1    2    3    4  . . . 16
  ----     --   --   --   --        --
  760539    !    2
  765130    !    1
  772744    !    !
  773268    !    1
  777055    !    !
 1180970    1    !
 1184873    !    !
 1932758    1    !
 1935343    1    !    !
 1935956    1    !    3
 1936699    !    !    3
 1937225    1    !    3
 1940973    !    !    3
 1941477    !    !    1
 1942046    4    !    1
 1945406    4    !    !
 1946282    4    !    1
 1946966    5    !    1
 1950431    5    !    !
 1951363    5    !    1
 1952095    6    !    1
 1954909    6    !    !
 1955655    6    !    7
 1959872    6    !    8
 1964010    6    !    9
 1968225    6    !    1
 2371705    6    !    1    !
60212892    6    !    !    !
60213188    6    !    ;    !
60313488    6    !    .    !
60355394    6    !    .    ;
60455675    6    !    .    .
60498138    6    ;    .    .
60598434    6    .    .    .
60646811    1    .    .    .


Key to symbols used
-------------------
  n  executing task n
  !  active
  .  inactive
  ;  in hold loop
```

8.3.1.4 User task status

FORMAT=STATUS, INTERVAL=2000000, in the COS JCL statement or -f status -i 2000000 in the UNICOS command line yields the status of up to 16 user tasks in uniform time intervals. Here, the interval equals 2,000,000 clock periods (default is 1,000,000).

```
          <---------- Internal Task Identifier ---------->
    Time   1    2    3    4    5    6    7    8    9 . . . 16
    ----  --   --   --   --   --   --   --   --   --       --
       0   !    *    !    !
 1000000   !         *    *    *    *    *    *    *
 2000000   .                        *
 3000000   !                        *
 4000000   !                        *
 5000000   !                        *
 6000000   !                        *
 7000000   !                        *
 8000000   !                        *
 9000000   !                        *
10000000   !                        *
11000000   !                        *
12000000   !                        *
13000000   !                        *
14000000   !                        *
15000000   !                        *
16000000   !                        *
17000000   !                        *
18000000   !                        *
19000000   !                        *
20000000   !                        *
21000000   !                        *
22000000   !                        *
23000000   !                        *
24000000   !                        *
25000000   !                        *
26000000   !                        *
27000000   !                        *
28000000   !                        *
29000000   !                        *
30000000   !                        *
31000000   !                        *
32000000   !                        *
33000000   !                        *
34000000   !                        *
35000000   !                        *
36000000   !                        *
37000000   !                        *
38000000   !                        *
39000000   !                        *
```

```
          <---------- Internal Task Identifier ----------->
    Time    1    2    3    4    5    6    7    8    9 . . . 16
    ----   --   --   --   --   --   --   --   --   --       --
40000000    !                        *
41000000    !                        *
42000000    !                        *
43000000    !                        *
44000000    !                        *
45000000    !                        *
46000000    !                        *
47000000    !                        *
48000000    !                        *
49000000    !                        *
50000000    !                        *
51000000    !                        *
52000000    !                        *
53000000    !                        *
54000000    !                        *
55000000    !                        *
56000000    !                        *
57000000    !                        *
58000000    !                        *
59000000    !                        *
60000000    *                        *
```

```
 Key to symbols used
 -------------------
  *  running
  !  ready; waiting for CPU
  .  waiting for lock, event, or task
```

In this example, seven tasks are shown running during time 1000000. Only four, of course, could be running at any one time with just four processors. The output suggests that there was a changeover in the tasks executing during that interval. Tasks 3 and 4 and probably one or two other tasks were running when the interval began, and task 6 and two or three other tasks were running when it ended.

8.3.1.5 Summary display

FORMAT=TOTALS in the COS JCL statement or -f totals in the UNICOS command line outputs the following summary statistics:

```
 Task activities
          9 TSKSTARTs
          9 task completions
          8 TSKWAITs
                  6 did not wait     ( 75.0%)
                  2 required a wait  ( 25.0%)
```

```
Lock variables
          1 locks encountered
          1 LOCKASGNs
          0 LOCKRELs
         10 LOCKONs
                10 did not wait      (100.0%)
                 0 required a wait   (  0.0%)
         10 LOCKOFFs

Event variables
          1 events encountered
          1 EVASGNs
          0 EVRELs
          1 EVWAITs
                 0 did not wait      (  0.0%)
                 1 required a wait   (100.0%)
          2 EVPOSTs
          2 EVCLEARs

Library scheduler:  logical CPU connections
         20 attaches
         12 detaches
          3 CPUs acquired
          3 spin-waits

Library scheduler:  system requests
          3 requests
          0 deletes
          3 suspends
          0 activates

No user defined actions found

Detailed task wait statistics

   TSKWAITs for user task: 00000000002
    Total TSKWAITS:          1
       no wait required:           1
          wait required:           0

   TSKWAITs for user task: 00000000003
    Total TSKWAITS:          1
       no wait required:           1
          wait required:           0

   TSKWAITs for user task: 00000000004
    Total TSKWAITS:          1
       no wait required:           1
          wait required:           0
```

```
    TSKWAITs for user task: 00000000005
     Total TSKWAITS:          1
        no wait required:          1
           wait required:          0

    TSKWAITs for user task: 00000000006
     Total TSKWAITS:          1
        no wait required:          0
           wait required:          1
          avg. wait time:                58677301 clocks

    TSKWAITs for user task: 00000000007
     Total TSKWAITS:          1
        no wait required:          1
           wait required:          0

    TSKWAITs for user task: 00000000010
     Total TSKWAITS:          1
        no wait required:          1
           wait required:          0

    TSKWAITs for user task: 00000000011
     Total TSKWAITS:          1
        no wait required:          0
           wait required:          1
         avg. wait time:                    14235 clocks

Detailed lock statistics

    Lock variable at address: 00000054413
           LOCKASGNs:           1
            LOCKRELs:           0
             LOCKONs:          10
            no wait required:         10  (100.0%)
               wait required:          0  (  0.0%)
            LOCKOFFs:          10

 Detailed event statistics

    Event variable at address: 00000054414
             EVASGNs:           1
              EVRELs:           0
             EVPOSTs:           2
             EVWAITs:           1
            no wait required:          0  (  0.0%)
               wait required:          1  (100.0%)
              avg. wait time:                   7909 clocks
            EVCLEARs:           2
```

8.3.2 EVENTS PARAMETER

This COS JCL statement or UNICOS command line yields an output of only the actions that involve events.

```
MTDUMP,DN=dumpdn,EVENTS.
mtdump -E dumpfile
```

Time	Task	Action	Action-dependent data
1621	1	assign event	00047741
359183	1	begin wait for event	00047741
360640	2	post event	00047741
362522	2	clear event	00047741
362680	1	run after wait for event	00047741
363092	1	post event	00047741
363438	1	clear event	00047741

8.3.3 TASKS PARAMETER

This COS JCL statement or UNICOS command line yields an output of only the actions that involve tasks; these include task start and completion, task waits, and task test.

```
MTDUMP,DN=dumpdn,TASKS.
mtdump -T dumpfile
```

Time	Task	Action	Action-dependent data
0	1	start task	00000000000000000000
4363	2	start task	00000000000000000002
1535837	2	complete task	00000000000000000002
1553992	1	TSKWAIT, no wait	2
1556783	3	start task	00000000000000000003
1560909	4	start task	00000000000000000004
1951310	5	start task	00000000000000000005
2378672	6	start task	00000000000000000006
2382796	7	start task	00000000000000000007
2386879	8	start task	00000000000000000010
2390960	9	start task	00000000000000000011
16355452	4	complete task	00000000000000000004
16411316	1	begin wait for task	3
16415279	5	complete task	00000000000000000005
16526420	3	complete task	00000000000000000003
30435080	6	complete task	00000000000000000006
30437023	1	run after wait for task	3
30437556	1	TSKWAIT, no wait	4
30438065	1	TSKWAIT, no wait	5
30438574	1	TSKWAIT, no wait	6

```
    Time  Task  Action                    Action-dependent data
    ----  ----  ------                    ---------------------
30439033    1   begin wait for task            7
30488255    7   complete task          00000000000000000007
30489225    1   run after wait for task        7
30489677    1   begin wait for task            8
30491617    8   complete task          00000000000000000010
30492593    1   run after wait for task        8
30493011    1   begin wait for task            9
30771250    9   complete task          00000000000000000011
30773152    1   run after wait for task        9
34066223    1   complete task          00000000000000000000
```

8.3.4 ACTION PARAMETER

This COS JCL statement or UNICOS command line lets you select the specific actions to be listed, using the numeric codes for the actions listed in subsection 8.2.1, BUFTUNE: Select Actions to be Recorded. The default is to list entries for all of the action codes; only the use of one or more of the ACTION, TASKS, LOCKS, EVENTS, CPUS, or USERS parameters overrides this. The four actions chosen in this example are as follows:

```
Action = 1  Start task
         2  Complete task
         4  Begin wait for task
         5  Run after wait for task
```

```
MTDUMP,DN=dumpdn,ACTION=1:2:4:5.
mtdump -a 1,2,4,5 dumpfile
```

```
    Time  Task  Action                    Action-dependent data
    ----  ----  ------                    ---------------------
       0    1   start task             00000000000000000000
    4363    2   start task             00000000000000000002
 1535837    2   complete task          00000000000000000002
 1556783    3   start task             00000000000000000003
 1560909    4   start task             00000000000000000004
 1951310    5   start task             00000000000000000005
 2378672    6   start task             00000000000000000006
 2382796    7   start task             00000000000000000007
 2386879    8   start task             00000000000000000010
 2390960    9   start task             00000000000000000011
16355452    4   complete task          00000000000000000004
16411316    1   begin wait for task            3
16415279    5   complete task          00000000000000000005
16526420    3   complete task          00000000000000000003
30435080    6   complete task          00000000000000000006
30437023    1   run after wait for task        3
```

```
    Time  Task  Action                    Action-dependent data
    ----  ----  ------                    ---------------------
30439033    1   begin wait for task            7
30488255    7   complete task          00000000000000000007
30489225    1   run after wait for task        7
30489677    1   begin wait for task            8
30491617    8   complete task          00000000000000000010
30492593    1   run after wait for task        8
30493011    1   begin wait for task            9
30771250    9   complete task          00000000000000000011
30773152    1   run after wait for task        9
34066223    1   complete task          00000000000000000000
```

8.3.5 TASK PARAMETER

This COS JCL statement or UNICOS command line produces a listing of internal task identifier numbers of the tasks recorded in the multitasking history trace buffer that you want to see. The default is all tasks.

```
MTDUMP,DN=dumpdn,TASK=4:5:6.
mtdump -t 4,5,6 dumpfile
```

```
    Time  Task  Action                    Action-dependent data
    ----  ----  ------                    ---------------------
 1560909    4   start task             00000000000000000004
 1950719    4   attach to logical CPU      3
 1951310    5   start task             00000000000000000005
 2378672    6   start task             00000000000000000006
 2380419    5   attach to logical CPU      1
16354735    4   set lock, no wait       00047740
16355097    4   clear lock              00047740
16355452    4   complete task          00000000000000000004
16357208    6   attach to logical CPU      3
16414562    5   set lock, no wait       00047740
16414924    5   clear lock              00047740
16415279    5   complete task          00000000000000000005
30434358    6   set lock, no wait       00047740
30434723    6   clear lock              00047740
30435080    6   complete task          00000000000000000006
```

8.3.6 DATA PARAMETER

This COS JCL statement or UNICOS command line lets you specify action-dependent data values to be searched for in the multitasking history trace buffer. The default is to list entries for all data values.

```
MTDUMP,DN=dumpdn,DATA=47741:3.
mtdump -d 47741,3 dumpfile
```

```
    Time  Task  Action                    Action-dependent data
    ----  ----  ------                    ---------------------
    1621     1  assign event               00047741
  359183     1  begin wait for event       00047741
  360640     2  post event                 00047741
  362522     2  clear event                00047741
  362680     1  run after wait for event   00047741
  363092     1  post event                 00047741
  363438     1  clear event                00047741
 1556783     3  start task                 00000000000000000003
 1950257        acquire logical CPU           3
 1950719     4  attach to logical CPU         3
16357208     6  attach to logical CPU         3
16411316     1  begin wait for task           3
16526420     3  complete task              00000000000000000003
30436717     1  attach to logical CPU         3
30437023     1  run after wait for task       3
30439544     1  detach from logical CPU       3
30439812        spin-wait logical CPU         3
30488914     1  attach to logical CPU         3
30490189     1  detach from logical CPU       3
30490689        spin-wait logical CPU         3
30492286     1  attach to logical CPU         3
30493563     1  detach from logical CPU       3
30494066        spin-wait logical CPU         3
30594344        suspend logical CPU           3
```

8.3.7 INFO KEYWORD

The following example shows two outputs, the first resulting from using BUFTUNE with the INFO keyword and calling BUFPRINT within the user program, and the second resulting from the JCL statement MTDUMP, the user having previously called BUFUSER to add entries to the multitasking history trace buffer. In this example, MTDUMP outputs the task calls and user calls.

The first output is dumped by calling BUFPRINT within the user program. The call to BUFTUNE with the DN keyword specifies the dataset to which the multitasking history trace buffer is to be dumped. The INFO keyword in the BUFTUNE call lets you associate descriptive (mnemonic) strings with the actions that you want recorded.

The second output is from the MTDUMP statement using the input buffer from the first output example. The MTDUMP JCL statement dumps the multitasking history trace buffer entries that were selected with the TASKS and USERS parameters.

```
MULTI.
CFT,ALLOC=STACK.
SEGLDR.
$ABD.
DUMPJOB.
MTDUMP,DN=dumpdn,TASKS,USERS.
/EOF
```

```
  PROGRAM TEST
  CHARACTER*24 STRING
  INTEGER ISTRING(3)
  DATA ISTRING/'THIS IS STRING NO. 66'H/
  . . .
  CALL BUFTUNE('DN'H,'dumpdn'H)
  STRING = 'THIS IS STRING NO. 65'
  CALL BUFTUNE('INFO'H,65,STRING)
  CALL BUFTUNE('INFO'H,66,ISTRING)
  CALL BUFTUNE('INFO'H,67,'THIS IS STRING NO. 67'H)
  CALL BUFTUNE('INFO'H,68,'THIS IS STRING NO. 68'H)
  CALL BUFTUNE('INFO'H,69,'THIS IS STRING NO. 69'H)
  CALL BUFTUNE('INFO'H,128,'THIS IS STRING NO.128'H)
  . . .
  IEMPTY = 0
  CALL BUFPRINT(IEMPTY)
  . . .
  END
  . . .
  SUBROUTINE SUB2
  C       task
  . . .
  CALL TSKVALUE(ITASK)
  CALL BUFUSER(65,ITASK)
  CALL BUFUSER(66,'DATA 66'H)
  CALL BUFUSER(67,'DATA 67'H)
  CALL BUFUSER(68,'DATA 68'H)
  CALL BUFUSER(69,'DATA 69'H)
  CALL BUFUSER(69,777B)
  CALL BUFUSER(70,'DATA 70'H)
  CALL BUFUSER(71,'DATA 71'H)
  CALL BUFUSER(128,'DATA 128'H)
  CALL BUFUSER(129,'DATA 129'H)
  CALL BUFUSER(130,130B)
  . . .
  END
```

Time	Task	Action	Action-dependent data	
0	1	start task	00000000000000000000	
4688	1	assign lock	00053240	
5040	1	assign event	00053241	
7948	2	start task	00000000000000000002	
8789	1	detach from logical CPU	0	
9029		request logical CPU		
775900		acquire logical CPU	2	
776375	1	attach to logical CPU	2	
776895	1	begin wait for event	00053241	
777418	1	detach from logical CPU	2	
777904	2	attach to logical CPU	1	
778177		spin-wait logical CPU	2	
778384	2	post event	00053241	
779148	2	detach from logical CPU	1	
779663	2	attach to logical CPU	2	
780214	1	attach to logical CPU	1	
780394	2	clear event	00053241	
780578	1	run after wait for event	00053241	
781037	1	post event	00053241	
781418	1	clear event	00053241	
3044299	1	set lock, no wait	00053240	
3044731	1	clear lock	00053240	
3045246	1	begin wait for task	2	
3045846	1	detach from logical CPU	1	
3046154		spin-wait logical CPU	1	
3146412		suspend logical CPU	1	
132777326	2	set lock, no wait	00053240	
132777729	2	clear lock	00053240	
132778079	2	complete task	00000000000000000002	
132779742	1	attach to logical CPU	2	
132780064	1	run after wait for task	2	
132782953	3	start task	00000000000000000003	
132783768	1	detach from logical CPU	2	
132784123		activate logical CPU	1	
133076653	1	attach to logical CPU	1	
133079629	4	start task	00000000000000000004	
133080442	1	detach from logical CPU	1	
133080682		request logical CPU		
133918469		acquire logical CPU	3	
133918932	1	attach to logical CPU	3	
133920086	3	attach to logical CPU	2	
133920627	3	THIS IS STRING NO. 65	00000000000000000003	
133920925	3	THIS IS STRING NO. 66	0421012504044015433040	DATA 66
133921200	3	THIS IS STRING NO. 67	0421012504044015433440	DATA 67
133921341	4	attach to logical CPU	1	
133921479	3	THIS IS STRING NO. 68	0421012504044015434040	DATA 68
133921754	3	THIS IS STRING NO. 69	0421012504044015434440	DATA 69

```
     Time  Task  Action                  Action-dependent data
     ----  ----  ------                  ---------------------
133921885    4   THIS IS STRING NO. 65   00000000000000000004
133922024    3   THIS IS STRING NO. 69   00000000000000000777
133922187    5   start task              00000000000000000005
133922309    3          70               042101250404401563004O   DATA 70
133922415    4   THIS IS STRING NO. 66   042101250404401543304O   DATA 66
133922594    3          71               042101250404401563044O   DATA 71
133922689    4   THIS IS STRING NO. 67   042101250404401543344O   DATA 67
133922851    3   THIS IS STRING NO.128   042101250404401423107O   DATA 128
133922970    4   THIS IS STRING NO. 68   042101250404401543404O   DATA 68
133923113    1   detach from logical CPU      3
133923221    3   00000000000000000201    042101250404401423107l   DATA 129
133923359        request logical CPU
133923447    4   THIS IS STRING NO. 69   042101250404401543444O   DATA 69
133923542    3   00000000000000000202    00000000000000000130
133923707    4   THIS IS STRING NO. 69   00000000000000000777
133923964    4          70               042101250404401563004O   DATA 70
133924228    4          71               042101250404401563044O   DATA 71
133924491    4   THIS IS STRING NO.128   042101250404401423107O   DATA 128
133924735    4   00000000000000000201    042101250404401423107l   DATA 129
133924972    4   00000000000000000202    00000000000000000130
248569711        acquire logical CPU          4
248570171    1   attach to logical CPU        4
248571265    5   attach to logical CPU        3
248571796    5   THIS IS STRING NO. 65   00000000000000000005
    .
    .
    .
455841660        suspend logical CPU          4
467556467    9   set lock, no wait        00053240
467556864    9   clear lock               00053240
467557229    9   complete task           00000000000000000011
467558554        spin-wait logical CPU        3
467658846        suspend logical CPU          3
471358042    6   set lock, no wait        00053240
471358451    6   clear lock               00053240
471358798    6   complete task           00000000000000000006
471360523    1   attach to logical CPU        2
471360844    1   run after wait for task      6
471361261    1   begin wait for task          7
471361800    1   detach from logical CPU      2
471362092        spin-wait logical CPU        2
471462336        suspend logical CPU          2
477017939    7   set lock, no wait        00053240
477018349    7   clear lock               00053240
477018704    7   complete task           00000000000000000007
477020408    1   attach to logical CPU        1
477020731    1   run after wait for task      7
```

MULTITASKING HISTORY TRACE BUFFER BUFPRINT

Time	Task	Action	Action-dependent data
477021187	1	TSKWAIT, no wait	8
477021653	1	TSKWAIT, no wait	9
477022205	1	assign lock	00003560
477022611	1	set lock, no wait	00003560

MULTITASKING HISTORY TRACE BUFFER MTDUMP

Time	Task	Action	Action-dependent data	
0	1	start task	00000000000000000000	
7948	2	start task	00000000000000000002	
3045246	1	begin wait for task	2	
132778079	2	complete task	00000000000000000002	
132780064	1	run after wait for task	2	
132782953	3	start task	00000000000000000003	
133079629	4	start task	00000000000000000004	
133920627	3	THIS IS STRING NO. 65	00000000000000000003	
133920925	3	THIS IS STRING NO. 66	042101250404401543304O	DATA 66
133921200	3	THIS IS STRING NO. 67	042101250404401543344O	DATA 67
133921479	3	THIS IS STRING NO. 68	042101250404401543404O	DATA 68
133921754	3	THIS IS STRING NO. 69	042101250404401543444O	DATA 69
133921885	4	THIS IS STRING NO. 65	00000000000000000004	
.				
.				
.				
255246551	7	THIS IS STRING NO. 66	042101250404401543304O	DATA 66
255246808	7	THIS IS STRING NO. 67	042101250404401543344O	DATA 67
255247081	7	THIS IS STRING NO. 68	042101250404401543404O	DATA 68
255247338	7	THIS IS STRING NO. 69	042101250404401543444O	DATA 69
255247595	7	THIS IS STRING NO. 69	00000000000000000777	
255247852	7	70	042101250404401563004O	DATA 70
255248109	7	71	042101250404401563044O	DATA 71
255248366	7	THIS IS STRING NO.128	042101250404401423107O	DATA 128
439079590	1	TSKWAIT, no wait	3	
439080320	1	TSKWAIT, no wait	4	
439080957	1	begin wait for task	5	
439082622	8	THIS IS STRING NO. 65	00000000000000000010	
439082903	8	THIS IS STRING NO. 66	042101250404401543304O	DATA 66
439083172	8	THIS IS STRING NO. 67	042101250404401543344O	DATA 67
439083442	8	THIS IS STRING NO. 68	042101250404401543404O	DATA 68
439083706	8	THIS IS STRING NO. 69	042101250404401543444O	DATA 69
439084009	8	THIS IS STRING NO. 69	00000000000000000777	
439084283	8	70	042101250404401563004O	DATA 70
439084540	8	71	042101250404401563044O	DATA 71
439084797	8	THIS IS STRING NO.128	042101250404401423107O	DATA 128

Time	Task	Action	Action-dependent data	
453550985	5	complete task	00000000000000000005	
453553406	9	THIS IS STRING NO. 65	00000000000000000011	
453553680	9	THIS IS STRING NO. 66	042101250404401543304O	DATA 66
453553938	9	THIS IS STRING NO. 67	042101250404401543344O	DATA 67
453554216	9	THIS IS STRING NO. 68	042101250404401543404O	DATA 68
453554490	9	THIS IS STRING NO. 69	042101250404401543444O	DATA 69
453554753	9	THIS IS STRING NO. 69	00000000000000000777	
453555010	9	70	042101250404401563004O	DATA 70
453555267	9	71	042101250404401563044O	DATA 71
453555527	9	THIS IS STRING NO.128	042101250404401423107O	DATA 128
455738083	8	complete task	00000000000000000010	
455740025	1	run after wait for task	5	
455740552	1	begin wait for task	6	
467557229	9	complete task	00000000000000000011	
471358798	6	complete task	00000000000000000006	
471360844	1	run after wait for task	6	
471361261	1	begin wait for task	7	
477018704	7	complete task	00000000000000000007	
477020731	1	run after wait for task	7	
477021187	1	TSKWAIT, no wait	8	
477021653	1	TSKWAIT, no wait	9	
483466811	1	complete task	00000000000000000000	

9. ADVANCED MACROTASKING IN FORTRAN

Several high-level concepts have emerged from the research on parallel processing. This section describes high-level parallelism concepts, both those that have been implemented directly through the macrotasking library routines and extensions to Fortran and those that can be implemented indirectly, using the macrotasking library routines.

9.1 PARALLELISM

Multitasking exploits parallelism in programs. Parallelism occurs when certain independence and order requirements are satisfied. The degree of parallelism is based on the adherence of program constructs to these requirements. The following levels of parallelism are discernable:

Level	Name	Characteristics
Full	Concurrent	Complete independence and order independence
Partial	Exclusive	Special dependence relationship and order independence
None	Sequential	Dependence and order dependence

Concurrent parallelism and exclusive parallelism are candidates for multitasking. No multitasking speedup is possible without parallelism.

The TSKSTART and TSKWAIT library routines initiate and complete concurrent parallelism. The initiation and completion of a task can be used as synchronization points.

The management of concurrent parallelism uses the EVPOST and EVWAIT library routines, which synchronize work between tasks and can be used in communicating data among tasks.

Exclusive parallelism is managed with the LOCKON and LOCKOFF library routines, which monitor special program segments that can be executed in any order, but not simultaneously. Exclusive parallelism is profitably exploited only within the environment of concurrent parallelism.

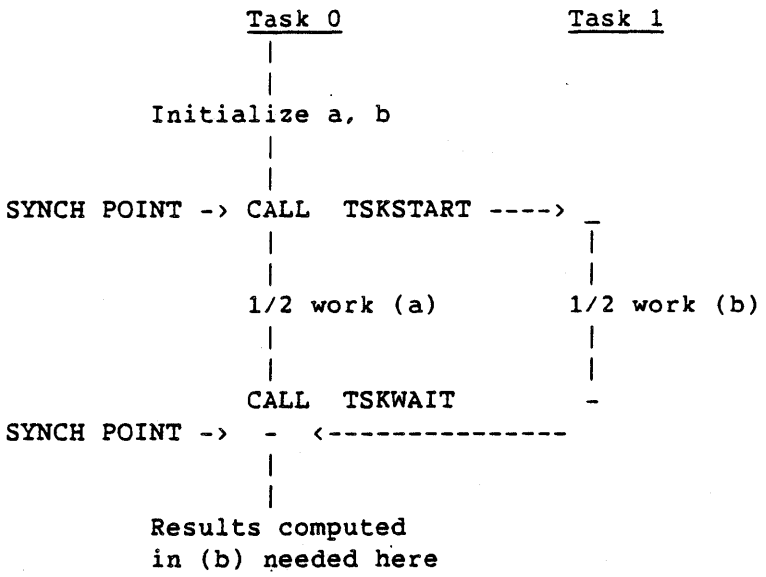
Subsequent subsections expand on these ideas.

9.2 SYNCHRONIZATION

Synchronization is the process of bringing two or more tasks to a known and coordinated stage in their execution. The location in each task where this happens is called a synchronization point. Synchronization is required to ensure that dependencies are satisfied, which frequently means ensuring that variables in one task are computed before they are used in another task.

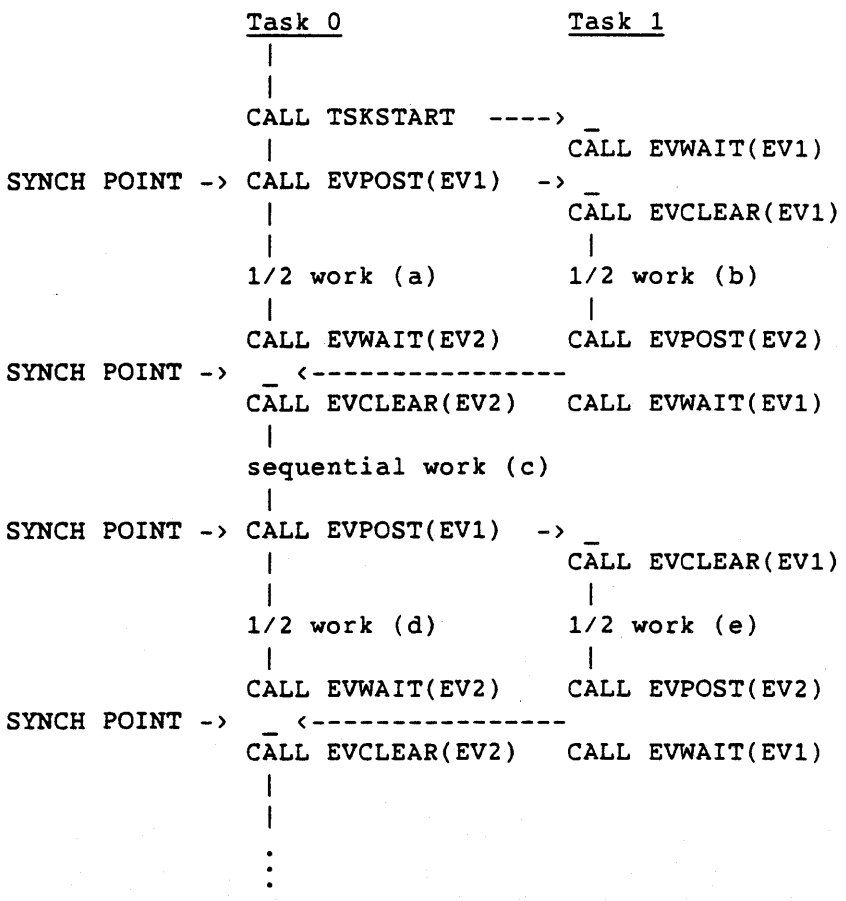
Synchronization is a cooperative process among tasks. Certain variables, such as event variables, must be accessible to all tasks, and each task must execute proper, coordinated multitasking calls.

The initiation and completion of a task are synchronization points, as the following example shows:

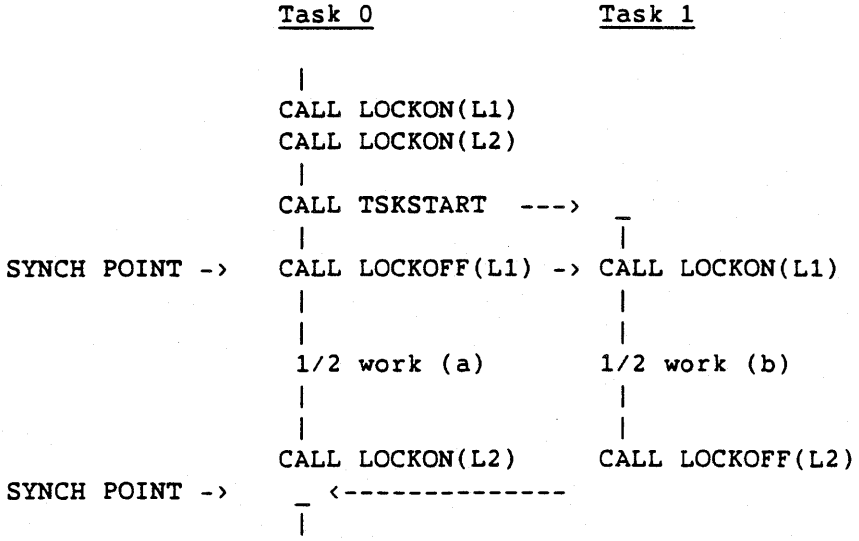


Management of concurrent parallelism uses events for synchronization. Tasks agree on which events signal the beginning and end of requested work.

In the following example, task 0 uses event EV1 to synchronize these two tasks by signaling task 1 that any initialization for work (b) is complete. Task 1 uses event EV1 to synchronize these two tasks by waiting for it to be posted before beginning work (b). In a similar fashion, both tasks use event EV2 to synchronize the completion of work (a) and (b) before the start of work (c).



An alternative method uses locks for synchronization. This technique requires that the locks be initialized to the locked state. Each synchronizing task needs only one call to the multitasking library.



```
                    Task 0                Task 1

                    sequential work (c)
                     |
    SYNCH POINT ->   CALL LOCKOFF(L1) -> CALL LOCKON(L1)
                     |                    |
                     |                    |
                    1/2 work (d)         1/2 work (e)
                     |                    |
                     |                    |
                    CALL LOCKON(L2)      CALL LOCKOFF(L2)
    SYNCH POINT ->   _ <--------------
                     |
                     |
```

Barriers can also be used to implement synchronization, as follows:

```
                    Task 0                Task 1

                    CALL BARASGN(B1,2)
                    CALL TSKSTART ----->  |
SYNCH POINT ->      CALL BARSYNC(B1)      CALL BARSYNC(B1)
                     |                    |
                     |                    |
                    1/2 work (a)         1/2 work (b)
                     |                    |
                     |                    |
SYNCH POINT ->      CALL BARSYNC(B1)<-->CALL BARSYNC(B1)
                     |
                     |
                    Sequential work (c)
                     |
                     |
SYNCH POINT ->      CALL BARSYNC(B1)<-->CALL BARSYNC(B1)
                     |                    |
                     |                    |
                    1/2 work (d)         1/2 work (e)
                     |                    |
                     |                    |
SYNCH POINT ->      CALL BARSYNC(B1)<-->CALL BARSYNC(B1)
                     |
                     |
```

Barriers are also convenient for synchronizing larger numbers of tasks.

9.3 COMMUNICATION

Occasionally one task must communicate a variable value to other tasks while all are executing. To ensure that the value is computed in one task before it is used in another, the communication must occur at a synchronization point. The communicating tasks must agree on the location of the shared value. One task computes the value before the synchronization point, and the other tasks reference the value only after the synchronization point.

In the following program, the main task uses the shared variable JOB to indicate the computations to be executed by the subordinate task, T. Task T stops when JOB equals 3.

```
      PROGRAM MAIN
      COMMON/MT/ISTART,IDONE,JOB,A(1000),B(1000),C(1000)
      ...   call EVASGN, etc.
      CALL TSKSTART(IDTASK,T)
      JOB = 1
      CALL EVPOST(ISTART)
      DO 10 I = 1, 500
        B(I) = A(I) + 1.0
 10   CONTINUE
      CALL EVWAIT(IDONE)
      CALL EVCLEAR(IDONE)
      JOB = 2
      CALL EVPOST(ISTART)
      DO 20 I = 501, 1000
        C(I) = B(I) + 2.0
 20   CONTINUE
      CALL EVWAIT(IDONE)
      CALL EVCLEAR(IDONE)
      JOB = 3
      CALL EVPOST(ISTART)
      CALL TSKWAIT(IDTASK)
      STOP
      END

      SUBROUTINE T
      COMMON/MT/ISTART,IDONE,JOB,A(1000),B(1000),C(1000)
  1   CALL EVWAIT(ISTART)
      CALL EVCLEAR(ISTART)
      IF( JOB.EQ.2 ) GO TO 19
      IF( JOB.GT.2 ) GO TO 99
      DO 10 I = 501, 1000
        B(I) = A(I) + 1.0
 10   CONTINUE
      CALL EVPOST(IDONE)
      GO TO 1
 19   CONTINUE
```

```
      DO 20 I = 1, 500
        C(I) = B(I) + 2.0
 20   CONTINUE
      CALL EVPOST(IDONE)
      GO TO 1
 99   CONTINUE
      RETURN
      END
```

The integrity of the variable JOB is ensured because the programmer has defined and followed a rule allowing the main task to reference JOB only after IDONE is posted and before ISTART is posted, and allowing task T to reference JOB only after ISTART is posted and before IDONE is posted.

9.4 MONITOR

Certain program constructs have data and storage dependencies that at first appear to prevent parallel processing. These constructs involve updating a variable using an operation that is both commutative and associative, such as addition or multiplication. This is the case of exclusive parallelism, and it can be executed in parallel if you know any update of a variable will never interfere with any other update of that same variable. You can be certain that this is the case if you monitor the updates to ensure that only one update is ever executing at a given time.

```
      DO 20 I = 1, N
        DO 10 J = 1, N
          A(I,J) = B(I)*C(J)
 10     CONTINUE
        S = S + SIN(A(I,1))
 20   CONTINUE
```

In the previous example, the iterations of the 20 loop are data- and storage-dependent because of the variable S. The dependence causes a problem if the updates to S are attempted in different tasks. Within a given task, fetching S may not obtain the correct value if the other task is currently in the update process. Simultaneous updates may overwrite and lose a needed value.

This problem can be circumvented and the iterations of the 20 loop executed in parallel if the updates are never simultaneous. Two solutions are possible:

- Use the LOCK facilities to form a critical region around the problem code. Subsection 9.10, Critical Region, describes this alternative in detail.

- Recognize that only the last value, rather than intermediate values, of S is required. Refer to subsection 9.11, Summation and Other Reduction Constructs.

9.5 SHARED AND PRIVATE VARIABLES

To ensure independence, you must analyze the use of variables in the code you intend to multitask. Based on the analysis results, you can deliberately allocate the variables according to their use.

The allocation of variables in the original program may conflict with their multitasked use. The modifications that you make for multitasking can affect portions of the program that are not being multitasked. The allocation of variables for use in multitasking is one of the most important steps in conversion, and it is too easily overlooked. Failure to address this aspect of multitasking gives rise to subtle data dependency violations that are difficult to identify during debugging.

You can categorize variables used in a multitasked segment of code according to the way they are used by the tasks that have access to them. In a correctly multitasked code segment, you can categorize variables as one of the following types:

Type	Category
1	Shared, read-only access
2	Shared, partitioned access
3	Shared, protected access
4	Private, with no restrictions on use but always defined before use

The first three variable types are accessible to all tasks, while the fourth is accessible to only one task.

The multitasked use of a variable is a characterization of the variable's accessibility by tasks. A variable's uniprocessing use is determined by the following:

- Its appearance on the left or right side of assignment statements
- Whether the variable is subscripted by the loop variable
- The variable's scope with respect to the loop

Variables within a multitasked loop can be characterized by their use within the loop as follows (the variable I is the loop index):

Type 1: Appears only on the right-hand side
May or may not be subscripted by I
Read-only references

```
      DO 1 I = 1, N
          ...  = A(I)
          ...  = S
    1 CONTINUE
```

Type 2: Appears on both left- and right-hand sides
Subscripted by I
Each iteration operates on independent elements

```
      DO 2 I = 1, N
      ...  = A(I)
      A(I) =   ...
      A(I) = A(I)
    2  CONTINUE
```

Type 3: Appears on the right- and left-hand side in one statement
Is not subscripted by I
Accumulation operation

```
      DO 3 I = 1, N
        S  =  S + ...
    3  CONTINUE
```

Type 4: Appears on the left-hand side, then on the right-hand side
Is not subscripted by I
Reused temporary variable

```
      DO 4 I = 1, N
        S  =  ...
        ...  =  S
    4  CONTINUE
```

Other uses of variables in loops inhibit multitasking.

The uniprocessing scope of variables with respect to the loop to be multitasked is also important. See subsection 2.6, Scope.

You must look at the original code and analyze the ramifications of introducing a new scope boundary in order to categorize the variables in the program and identify the allocation required for multitasking.

Consider the following original code:

```
      B = ...
      C = ...
      DO 10 I = 1, 1000
        A = B + FLOAT(I)
        C = C + 1.
        D(I) = A*2.
 10   CONTINUE
      ... = C
      ... = D(K)
```

The analysis showing that the iterations of the 10 loop can be done in parallel relies on the analysis of the multitasking use of the variables referenced in the loop. An attempt to multitask this loop introduces a new scope boundary at DO 10 and CONTINUE. Identify and compare the scope of the variables referenced in the loop with this new scope region. If the scope of a variable is contained within the introduced scope boundaries, that variable is private to each task. Private variables are always defined (assigned) before being used within the new scope boundaries. The variable A in the preceding example is a private variable.

If the scope of a variable extends outside the new scope boundaries, the variable is shared by all tasks. Program modifications are needed to maintain the variable's scope over the new multitasked code. Usually these modifications involve putting the variable into COMMON statements or putting the variable into the argument list of the TSKSTART statement. Avoid the latter method whenever possible, however, because passing arguments through TSKSTART causes many of the bugs in multitasked programs. The variables B, C, and D in the preceding example are shared variables. Variable B is a shared variable that has only fetch references, variable C is a shared variable that must be monitored to avoid simultaneous updates, and variable D is a COMMON variable whose elements are independently assigned.

You can convert the previous original code for multitasking as follows:

```
      MULTITASKED CODE
      COMMON / MT / LOCKC, B, C, D(1000)
      B = ...
      C = ...
      CALL TSKSTART( IDTASK, TASK )
      DO 10 I = 1, 500
        A = B*FLOAT(I)
        CALL LOCKON( LOCKC )
        C = C + 1.
        CALL LOCKOFF( LOCKC )
        D(I) = A*2.
 10   CONTINUE
```

```
      CALL TSKWAIT( IDTASK )
      ... = C
      ... = D(K)

      SUBROUTINE TASK
      COMMON / MT / LOCKC, B, C, D(1000)
      DO 10 I = 501, 1000
        A = B*FLOAT(I)
        CALL LOCKON( LOCKC )
        C = C + 1.
        CALL LOCKOFF( LOCKC )
        D(I) = A*2.
10    CONTINUE
      RETURN
      END
```

Two variables are now named A, one in MAIN and one in TASK. Storage location B is only fetched in both tasks and needs only to be made accessible to all tasks. Storage location C is both fetched and assigned and needs to be monitored as well as made accessible to all tasks. Different storage locations of array D are assigned by each task, but the whole array is accessible to all tasks.

The modifications for multitasking (putting variables into COMMON blocks) may interfere with the storage assignment of variables in the original program. In the preceding example, variable A may have been originally contained in a COMMON block. Its uniprocessing use accommodated its accessibility (or reusability) by several program units. Its multitasking use requires that it be private.

Likewise, variables B, C, and D may have been contained in COMMON blocks along with other variables not involved in multitasking. Also, the placement of these variables into COMMON blocks may interfere with the use of the variables or with other variables having the same name in other parts of the program. You must understand completely, over the whole program, the use of all variables involved in multitasking (and other variables with the same names). You must pay special attention to variables that are Fortran equivalenced to other variables.

The following rules aid in determining the categories of the variables appearing within a loop considered for multitasking. The loop control variable is assumed to be I.

1. Variable is subscripted by I - SHARED.

2. Variable is not subscripted by I.

    a. Variable appears only on the left-hand side; PRIVATE. Watch out for a live variable after loop.
    b. Variable appears only on the right-hand side; SHARED.
    c. Variable appears on both left- and right-hand sides

        1) Variable always defined before used; PRIVATE.
        2) Variable not defined before used; ERROR or SHARED (and monitored).

The characteristics of private and shared variables are important when identifying the multitasked use of variables:

- Variables private to tasks:

    - Multiple copies (one per task)
    - Temporary existence (dies when task dies)
    - Cannot be referenced by other tasks
    - Always defined before used within task
    - Usually scalars or small workspace arrays

- Variables shared by tasks:

    - One copy (independent of number of tasks)
    - Permanent existence (dies when job dies)
    - Can be referenced by all tasks (common)
    - Are fetch only, independently used, or monitored
    - Usually larger arrays, lock or event variables, constants

9.6 TASK COMMON

Standard Fortran provides COMMON blocks for sharing variables among program units. Common blocks are also used to minimize the length of argument lists in CALL statements.

Variables appearing in COMMON blocks occupy static storage locations for the life of the program. The fact that variables in COMMON blocks are accessible to all tasks and all program units can cause a conflict. A task might require that the different program units composing it all have access to a certain variable, therefore needing that variable in COMMON storage. The same task may also require, however, that no other tasks have access to the variable, necessitating that the variable be in private storage. Further, several different tasks can execute some of the same program units, requiring that you consider not only which program unit may access a variable but also which task a program unit is in when it accesses a variable.

TASK COMMON is a Cray Fortran language extension that provides a way for tasks to obtain storage that is private to the task but that can be used by all subroutines called within the task. The syntax for a TASK COMMON declaration is as follows:

```
TASK COMMON /cbname/nlist
```

TASK COMMON must be named, and the items cannot be saved or preset with DATA or NAMELIST I/O, but otherwise the syntax and usage are identical to those of regular COMMON.

When compiled with static storage allocation (the default) or the CFT77 STATIC option, TASK COMMON is treated in the same way as regular COMMON. When compiled with the STACK option, TASK COMMON blocks are set up on the stack when a task is started and go away when a task completes.

The following program illustrates a problem for which TASK COMMON provides a solution:

```
      COMMON / ARGS / A(100), B(100)
      COMMON / RESULT / C(100)
      DO 20 I = 1, 100
        DO 10 J = 1, 100
          A(J) = I+J
          B(J) = I*J
10      CONTINUE
        CALL SUB(I)
20    CONTINUE
      STOP
      END
      SUBROUTINE SUB(I)
      COMMON / ARGS / A(100), B(100)
      COMMON / RESULT / C(100)
      C(I) = SDOT(100,A,1,B,1)
      RETURN
      END
```

Without using TASK COMMON, this program can be converted for multitasking by duplicating code and using a variable, IOFFSET, to share the statically allocated COMMON block. This process must be followed for each task that is used, as in the following example:

```
      COMMON / ARGS / A(200), B(200)
      COMMON / RESULT / C(100)
      CALL TSKSTART( IDTASK, TASK )
      IOFFSET = 0
```

```
      DO 20 I = 1, 50
        DO 10 J = 1, 100
          A(J+IOFFSET) = I+J
          B(J+IOFFSET) = I*J
10      CONTINUE
        CALL SUB(I,IOFFSET)
20    CONTINUE
      CALL TSKWAIT( IDTASK )
      STOP
      END
      SUBROUTINE TASK
      COMMON / ARGS / A(200), B(200)
      COMMON / RESULT / C(100)
      IOFFSET = 100
      DO 20 I = 51, 100
        DO 10 J = 1, 100
          A(J+IOFFSET) = I+J
          B(J+IOFFSET) = I*J
10      CONTINUE
        CALL SUB(I,IOFFSET)
20    CONTINUE
      RETURN
      END
      SUBROUTINE SUB(I,IOFFSET)
      COMMON / ARGS / A(200), B(200)
      COMMON / RESULT / C(100)
      C(I) = SDOT(100,A(1+IOFFSET),1,B(1+IOFFSET),1)
      RETURN
      END
```

Alternatively, A and B can be made private variables in MAIN and TASK, and passed as arguments to SUB.

CFT77 and CFT contain a language extension that explicitly provides for a TASK COMMON capability. This extension permits a simple program modification to produce the desired results, as in the following example:

```
      TASK COMMON / ARGS / A(100), B(100)
      COMMON / RESULT / C(100)
      CALL TSKSTART( IDTASK, TASK )
      DO 20 I = 1, 50
        DO 10 J = 1, 100
          A(J) = I+J
          B(J) = I*J
10      CONTINUE
        CALL SUB(I)
20    CONTINUE
      CALL TSKWAIT( IDTASK )
      STOP
      END
```

```
      SUBROUTINE TASK
      TASK COMMON / ARGS / A(100), B(100)
      COMMON / RESULT / C(100)
      DO 20 I = 51, 100
        DO 10 J = 1, 100
          A(J) = I+J
          B(J) = I*J
 10     CONTINUE
        CALL SUB(I)
 20   CONTINUE
      RETURN
      END
      SUBROUTINE SUB(I)
      TASK COMMON / ARGS / A(100), B(100)
      COMMON / RESULT / C(100)
      C(I) = SDOT(100,A,1,B,1)
      RETURN
      END
```

Using TASK COMMON to provide a separate workspace for each task requires only that the COMMON /ARGS/ statement be replaced with TASK COMMON /ARGS/. Other program modifications may not be necessary. In the preceding example, two COMMON blocks have the name ARGS. One is accessible by the main task (main and SUB), while the second is accessible by the started task (TASK and SUB). References in subroutine SUB to A and B apply to the TASK COMMON block associated with the task to which SUB belongs.

9.7 DOALL

A DOALL is a loop with independent iterations. A partition of the iterations of a DO loop divides the iterations into groups. These groups can be executed in parallel by multiple tasks.

The partitioning of iterations is called static if the iterations belonging to each group are known before execution time. The partitioning of iterations is called dynamic if either the number of iterations belonging to a group or the assignment of iterations to groups is unknown until execution time.

For vectorization, most of the work is found in loops, which is where parallelism is exploited. This is also true of multitasking, but on a larger scale. A frequent application of multitasking is the simultaneous execution of independent iterations of loops that have been broken apart into separate subprograms. The techniques to multitask loops are of fundamental importance.

The first case considered is a loop in which each iteration has equal computational requirements. A choice exists as to how these independent iterations should be grouped for execution as distinct tasks. At one extreme, each iteration can be computed by a separate task. At the other extreme, the iterations can be grouped into a number of tasks equal to the number of processors. Having fewer groups (tasks) than processors prevents full use of the processor resources. In the middle, a balance of the number of groups and the number of iterations per group can enhance both vectorization and multitasking speedups in some cases.

For workloads with equal iterations and a given number of processors (for example, two), statically dividing the iterations into two groups is natural. Each group may comprise the even- or odd-numbered iterations or the first and second half. The even-odd partition may increase bank conflicts. Consider the following matrix addition example:

```
      DO 20 J = 1, N
        DO 10 I = 1, N
          A(I,J) = B(I,J) + C(I,J)
10      CONTINUE
20    CONTINUE
```

The 20 loop is multitasked by partitioning the range of J into first half and second half. This corresponds to partitioning the matrix addition problem into two parts: the left half of A and the right half of A. For a sufficiently large N, the right-half computation may be formed into a separate task, as follows:

Task 0

```
COMMON / MT / A,B,C,N
L = N/2
LP1 = L + 1
CALL TSKSTART(IDT,T,LP1,N)

CALL T(1,L)

CALL TSKWAIT(IDT)
```

Task 1

```
      SUBROUTINE T(IS,IE)
      COMMON / MT / A,B,C,N
      DO 20 J = IS, IE
        DO 10 I = 1, N
          A(I,J) = B(I,J)+C(I,J)
10      CONTINUE
20    CONTINUE
      RETURN
      END
```

The same subroutine T is part of both tasks in the preceding example. Its arguments determine which half of the computation is to be performed.

The overhead of task generation (for example, stack allocation of local variables) usually makes EVENTs better mechanisms for managing loop parallelism. This is especially true when several loops can be multitasked, as the following example shows:

```
      PROGRAM MAIN
      COMMON/MT/ISTART,IDONE,JOB,A(1000),B(1000),C(1000)
      CALL TSKSTART(IDTASK,T)
      JOB = 1
      CALL EVPOST(ISTART)
      DO 10 I = 1, 500
        B(I) = A(I) + 1.0
 10   CONTINUE
      CALL EVWAIT(IDONE)
      CALL EVCLEAR(IDONE)
      JOB = 2
      CALL EVPOST(ISTART)
      DO 20 I = 501, 1000
        C(I) = B(I) + 2.0
 20   CONTINUE
      CALL EVWAIT(IDONE)
      CALL EVCLEAR(IDONE)
      JOB = 3
      CALL EVPOST(ISTART)
      CALL TSKWAIT(IDTASK)
      STOP
      END

      SUBROUTINE T
      COMMON/MT/ISTART,IDONE,JOB,A(1000),B(1000),C(1000)
  1   CALL EVWAIT(ISTART)
      CALL EVCLEAR(ISTART)
      IF( JOB.EQ.2 ) GO TO 19
      IF( JOB.GT.2 ) GO TO 99
      DO 10 I = 501, 1000
        B(I) = A(I) + 1.0
 10   CONTINUE
      CALL EVPOST(IDONE)
      GO TO 1
 19   CONTINUE
      DO 20 I = 1, 500
        C(I) = B(I) + 2.0
 20   CONTINUE
      CALL EVPOST(IDONE)
      GO TO 1
 99   CONTINUE
      RETURN
      END
```

In the preceding example, the event ISTART signals the waiting task T to begin work specified by the flag JOB. The event IDONE signals the main task that work has been completed. Task T is programmed as an infinite loop always going back to statement 1 to look for more work to do. A flag value of 3 terminates task T.

When the workload of each iteration varies, a different technique may be appropriate. A static partition of the iterations may result in unequal execution times for each task, causing some tasks to wait unnecessarily for other tasks to complete. One solution is to have the iterations schedule themselves as follows:

Task 0

```
COMMON/ MT /A(100,100),J,JLOCK,N
J = 0
CALL TSKSTART(IDT,T)

CALL T

CALL TSKWAIT(IDT)
```

Task 1

```
      SUBROUTINE T
      COMMON/ MT /A(100,100),J,JLOCK,N
    1 CALL LOCKON( JLOCK )
      JLOCAL = J + 1
      J = JLOCAL
      CALL LOCKOFF( JLOCK )
      IF( JLOCAL.GT.N ) GO TO 99
      IF( A(1,JLOCAL).EQ.0.0 ) GO TO 20
      DO 10 I = 1, N
        A(I,JLOCAL) =
                A(I,JLOCAL)/A(1,JLOCAL)
   10 CONTINUE
   20 GO TO 1
   99 RETURN
      END
```

Subroutine T is a part of both tasks in the previous example. Each task accesses and updates a shared variable J, which is the outer loop variable. This access is under the protection of a lock to ensure exclusive updates. Each task copies the next value of J to a private location, JLOCAL, and commits itself to that iteration. When an iteration completes, each task goes back to look for unprocessed iterations until all are performed.

Which task computes a given iteration is unknown at the start. Using this technique, the workload tends to be balanced among the tasks. The task that commits to shorter iterations does more of them. The difference in completion time between the two tasks is, at most, one iteration time.

The dynamic partitioning technique incurs an overhead for each iteration, raising the question of whether the overhead compensates for the workload imbalance. An alternative is to dynamically schedule fixed size groups of iterations. A task then commits to a range of the values of J. This reduces the overhead, but it also reduces the capability for load balancing.

9.8 COBEGIN

A COBEGIN is a sequence of independent program segments, which may be loops or CALL statements. Because the segments are independent, profitable multitasking is possible if the segments are of similar size.

Example:

```
  CALL FFT( IN1, OUT1, TEMP1, N )
  CALL FFT( IN2, OUT2, TEMP2, N )
```

The independence of input, output, and workspace allows these Fast Fourier Transforms (FFTs) to be done in separate tasks. Multitasking speeds up the computation in this case, while vectorization across the FFTs does not help.

COBEGIN can be viewed as a generalization of DOALL with independent segments instead of independent iterations. The relationship is made clearer by transforming the segments into a loop.

```
      DO 10 I = 1, 2
        IF(I.EQ.1) CALL FFT( IN1, OUT1, TEMP1, N )
        IF(I.EQ.2) CALL FFT( IN2, OUT2, TEMP2, N )
  10  CONTINUE
```

This example is easily converted for multitasking.

```
      CALL TSKSTART( IDFFT, FFT, IN1, OUT1, TEMP1, N )
      CALL FFT( IN2, OUT2, TEMP2, N )
      CALL TSKWAIT( IDFFT )
```

9.9 DOPIPE

A DOPIPE is a software pipeline of program segments within a loop. Dependencies among the segments prevent the loop from having independent iterations. Nevertheless, the iterations of the loop can be executed in parallel if the dependencies are satisfied.

Example:

```
      DO 10 I = 2, N
        A(I) = A(I-1) + B(I)   <--- Statement 1
        D(I) = A(I) + C(I)     <--- Statement 2
  10  CONTINUE
```

In this example, dependencies involving the variable A prohibit performing the loop iterations independently. After the value is assigned to A in one iteration, however, the next iteration can begin. Figure 9-1 shows how the iterations are pipelined to satisfy the dependence requirement and to allow simultaneous execution of the iterations.

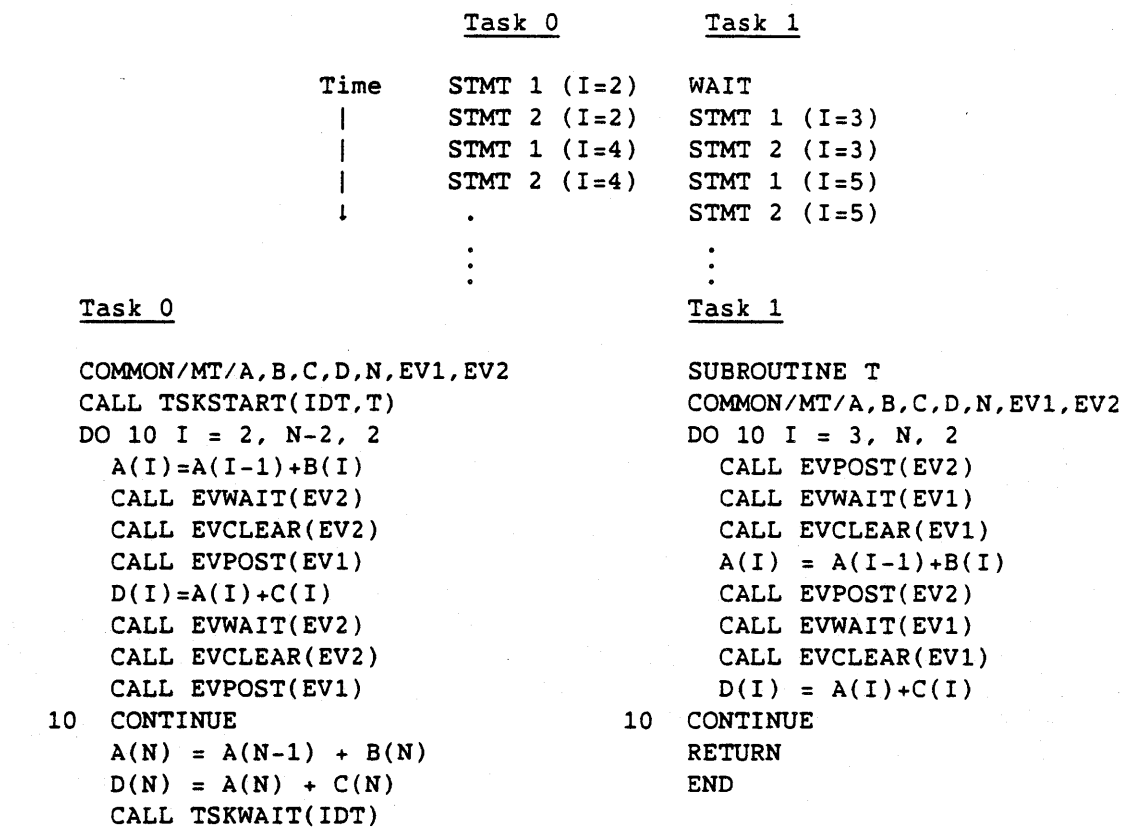


Figure 9-1. Pipelining

Use a dynamic pipeline to allow independent iterations, and use locks to monitor the commitment of tasks to iterations.

The following example gives an alternative, dynamic implementation:

Original code:

```
      DO 10 I=1,N
        A(I) = A(I-1) + B(I)
10      D(I) = A(I) + C(I)
```

Task 0

```
COMMON/ MT /A,B,C,D,N,I,L1
I = 1
CALL TSKSTART(IDT,T)

CALL T

CALL TSKWAIT(IDT)
```

Task 1

```
      SUBROUTINE T
      COMMON/ MT /A,B,C,D,N,I,L1
   10 CALL LOCKON(L1)
      IL = I+1
      I = IL
      IF( IL.GT.N ) GO TO 20
      A(IL) = A(IL-1) + B(IL)
      CALL LOCKOFF (L1)
      D(IL) = A(IL) + C(IL)
      GO TO 10
   20 CALL LOCKOFF(L1)
      RETURN
      END
```

The following example contains two (or more) piping segments. This implementation approach is independent of the number of segments or the number of processors.

Original code:

```
      DO 10 I=1,N
        A(I) = A(I-1) + B(I)
  10    D(I) = D(I-1) + A(I)
```

Task 0

```
COMMON/ MT /A,B,C,D,N,I,L1,L2
I = 1
CALL TSKSTART(IDT,T)

CALL T

CALL TSKWAIT(IDT)
```

Task 1

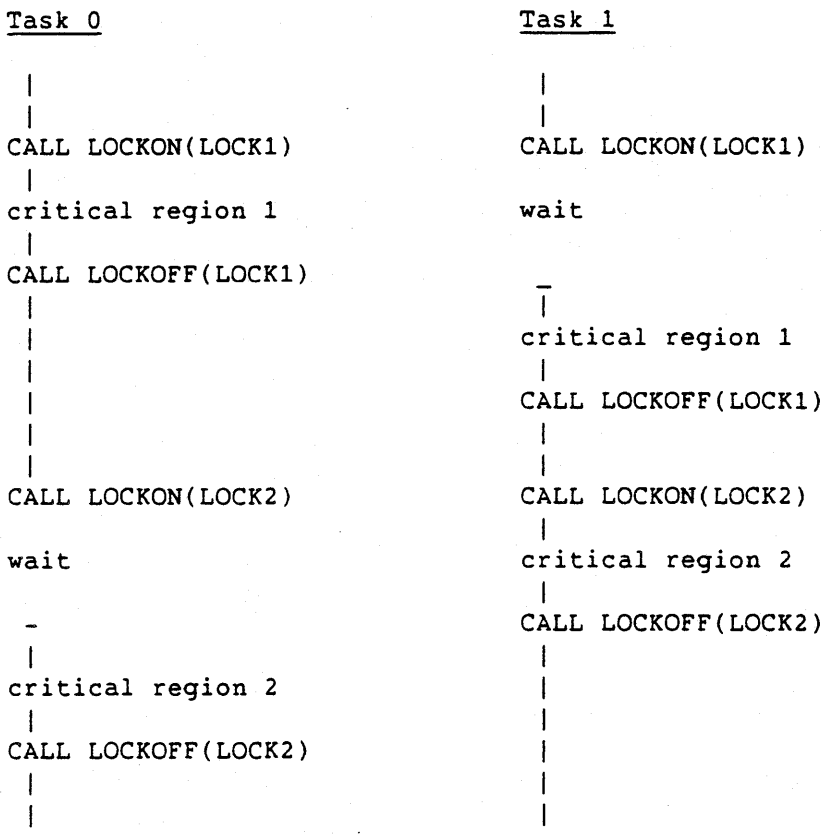
```
      SUBROUTINE T
      COMMON/ MT /A,B,C,D,N,I,L1,L2
   10 CALL LOCKON(L1)
      IL = I+1
      I = IL
      IF( IL.GT.N ) GO TO 20
      A(IL) = A(IL-1) + B(IL)
      CALL LOCKON (L2)
      CALL LOCKOFF (L1)
      D(I) = D(I-1) + A(I)
      CALL LOCKOFF(L2)
      GO TO 10
   20 CALL LOCKOFF(L1)
      RETURN
      END
```

For this technique to be worth the price of the overhead, the program segments in the pipeline must be of sufficient size. Additionally, each segment must be of similar size so that the workload for each task is balanced at every stage.

9.10 CRITICAL REGION

A critical region is a program segment that can be executed by only one task at a given time. The lock facilities ensure that this happens. Locks are not associated directly with any computational variable or its storage location; they are rather associated with code that references the variable.

A critical region is formed by turning the lock on before entering the program segment and off after leaving. A critical region may be one physical program segment or corresponding program segments in different tasks. A task attempting to enter a critical region that is occupied by another task must wait until the other task exits the region, as shown in the following example:



The following example shows the use of locks to manage exclusive parallelism in the environment of concurrent parallelism. Consider the original code segment:

```
      S = 0.0
      DO 20 I = 1, N
        DO 10 J = 1, N
          A(I,J) = B(I)*C(J)
10      CONTINUE
        S = S + SIN(A(I,1))
20    CONTINUE
```

The iterations of the 20 loop can be executed in parallel if the updates to S are never performed simultaneously. One lock protects the access of all tasks to this update statement. The multitasked code is as follows:

Task 0

```
      COMMON/MT/A,B,C,N,S,LOCKS
      CALL LOCKASGN( LOCKS )
      S = 0.0
      CALL TSKSTART( IDT, T )
      DO 20 I = 1, N/2
        DO 10 J = 1, N
          A(I,J) = B(I)*C(J)
10      CONTINUE
      CALL LOCKON( LOCKS )
      S = S + SIN(A(I,1))
      CALL LOCKOFF( LOCKS )
20     CONTINUE
      CALL TSKWAIT( IDT )
```

Task 1

```
      SUBROUTINE T
      COMMON/MT/A,B,C,N,S,LOCKS
      DO 20 I = N/2+1, N
        DO 10 J = 1, N
          A(I,J) = B(I)*C(J)
10      CONTINUE
        CALL LOCKON( LOCKS )
        S = S + SIN(A(I,1))
        CALL LOCKOFF( LOCKS )
20    CONTINUE
      RETURN
      END
```

The following subsection offers another approach to the multitasking of summation.

9.11 SUMMATION AND OTHER REDUCTION CONSTRUCTS

Occasionally, exclusive parallelism constructs are found within the context of otherwise completely independent code. Such constructs include summation, product, minimum, maximum, and search. One alternative for executing these constructs in parallel along with the rest of the code is presented in subsection 9.10, Critical Region. That technique uses locks to maintain the exclusive independence of the construct while exploiting a higher level of parallelism.

The technique presented here exploits parallelism in the constructs themselves. This technique is useful when parallelism is not present at a higher level or when these constructs form a significant portion of the work to be done.

Each task forms a partial result such as a partial sum or a local maximum, and the partial results are collected to form a total result by the main task after all tasks have synchronized. This approach does not require locks, because each task has independent storage locations in which to compute its partial result. The main task needs access, however, to all partial results. The following code shows the example of the previous subsection using this technique.

Task 0

```
      COMMON/MT/A,B,C,N,S1
      S0 = 0.
      S1 = 0.
      CALL TSKSTART( IDT, T )
      DO 20 I = 1, N/2
        DO 10 J = 1, N
          A(I,J) = B(I)*C(J)
10      CONTINUE
        S0 = S0 + SIN(A(I,1))
20    CONTINUE
      CALL TSKWAIT( IDT )
      S = S0 + S1
```

Task 1

```
      SUBROUTINE T
      COMMON/MT/A,B,C,N,S1
      DO 20 I = N/2+1, N
        DO 10 J = 1, N
          A(I,J) = B(I)*C(J)
10      CONTINUE
        S1 = S1 + SIN(A(I,1))
20    CONTINUE
      RETURN
      END
```

In the preceding code, task 0 computes its partial sum in S0, while task 1 computes its partial sum in S1. After task 1 completes, task 0 computes the total sum, S, from S0 and S1.

9.12 FORK/JOIN

You can implement FORK/JOIN in a Fortran program without moving code into a separate subroutine by setting up alternative entry points that TSKSTART calls indirectly. You must use a dummy subroutine because the compiler does not allow a subroutine to pass itself as an argument.

You must decide whether each variable is local or global, but this analysis aids in the understanding of the algorithm. Because the compiler must know how to handle each variable, each variable in the COMMON code is treated the same for each entry point. This means that all arguments to the subroutine must also be passed on to the alternative entry point so they are available to the COMMON code, and arguments that are passed only to the alternative entry point cannot be used by code entered directly through the main entry point.

A master/slave relationship must exist between the main task and the tasks that use the alternative entry point so that all of the subroutine's work is completed before it returns. The master task must always wait for all slave tasks to complete before returning.

The following examples show three ways to convert a simple matrix multiply routine (MXM is the routine from which the others are derived):

```
      SUBROUTINE MXM(A,L,B,M,C,N)
      REAL A(L,M),B(M,N),C(L,N)
C
      DO 40 I = 1,L
        DO 10 J = 1,N
          C(I,J) = 0
  10    CONTINUE
        DO 30 K = 1,M
          DO 20 J = 1,N
            C(I,J) = C(I,J) + A(I,K)*B(K,J)
  20      CONTINUE
  30    CONTINUE
  40  CONTINUE
C
      RETURN
      END
```

MULTMXM splits the matrix into two equal parts. TRANS is a simple way to avoid a recursive call.

```
      SUBROUTINE MULTMXM(A,L,B,M,C,N)    ! split into 2 equal tasks
      INTEGER TID(3)
      EXTERNAL TRANS
      REAL A(L,M),B(M,N),C(L,N)
C
      TID(1) = 2
      CALL TSKSTART(TID,TRANS,A,L,B,M,C,N)
C
C        Set up parameters for the master task.
C
      JL = L/2+1
      LL = L
      LCHILD = 0
      GOTO 1
C
C        Set up parameters for the slave task.
C
      ENTRY MULTMXM1(A,L,B,M,C,N)
      JL = 1
      LL = L/2
      LCHILD = 1
C
C        Common code for both tasks.
```

```
C
   1  CONTINUE
C
      DO 40 I = JL,LL
         DO 10 J = 1,N
            C(I,J) = 0
  10     CONTINUE
         DO 30 K = 1,M
            DO 20 J = 1,N
               C(I,J) = C(I,J) + A(I,K)*B(K,J)
  20        CONTINUE
  30     CONTINUE
  40  CONTINUE
C
C     The master task must wait for the slave to complete before
C     returning.
C
      IF (LCHILD .EQ. 0) CALL TSKWAIT(TID)
      RETURN
      END

      SUBROUTINE TRANS(A,L,B,M,C,N)          ! simple way to avoid a
      CALL MULTMXM1(A,L,B,M,C,N)             ! recursive call
      RETURN
      END
```

SELSYNDO uses a self-synchronizing DO-loop technique.

```
      SUBROUTINE SELSYNDO(A,L,B,M,C,N)       ! self-synchronizing DOs
      INTEGER TID(3)
      EXTERNAL TRANS1
      REAL A(L,M),B(M,N),C(L,N)
      DATA LOCKI,IGLOBAL /-1,0/
C
      CALL LOCKASGN(LOCKI,-1)
      IGLOBAL = 0
      TID(1) = 2
      CALL TSKSTART(TID,TRANS1,A,L,B,M,C,N)
C
C      Set the flag for the master task
C
      LCHILD = 0
      GOTO 1
C
C      Set up the slave task
C
      ENTRY SELSYND1(A,L,B,M,C,N)
      LCHILD = 1
C
C        Common code shared by both tasks
C
```

```
   1  CONTINUE
C
  99  CONTINUE
      CALL LOCKON (LOCKI)
         IGLOBAL = IGLOBAL + 1
         I = IGLOBAL
      CALL LOCKOFF(LOCKI)
      IF ( I .GT. L) GOTO 1000
      DO 10 J = 1,N
         C(I,J) = 0
  10  CONTINUE
      DO 30 K = 1,M
         DO 20 J = 1,N
            C(I,J) = C(I,J) + A(I,K)*B(K,J)
  20     CONTINUE
  30  CONTINUE
      GOTO 99
1000  CONTINUE
C
C        The master waits for the slave to complete before returning
C
      IF (LCHILD .EQ. 0) CALL TSKWAIT(TID)
      RETURN
      END

      SUBROUTINE TRANS1(A,L,B,M,C,N)
      CALL SELSYND1(A,L,B,M,C,N)
      END
```

SELSYNDN also uses the self-synchronizing DO-loop technique but is generalized for NPROC tasks.

```
      SUBROUTINE SELSYNDN(A,L,B,M,C,N)     ! self-synchronizing DOs
      PARAMETER (NPROC = 4)
      INTEGER TID(3,NPROC-1)
      EXTERNAL TRANS2
      REAL A(L,M),B(M,N),C(L,N)
      DATA LOCKI,IGLOBAL /-1,0/
C
      CALL LOCKASGN(LOCKI,-1)
      IGLOBAL = 0
C
      DO 2 I = 1,NPROC-1
         TID(1,I) = 3
         TID(3,I) = I
         CALL TSKSTART(TID(1,I),TRANS2,A,L,B,M,C,N)
 2    CONTINUE
C
C        Set up the flag for the master task
C
      LCHILD = 0
```

```
      GOTO 1
C
C     Set up the flags for the slave tasks
C
      ENTRY SELSYND2(A,L,B,M,C,N)
      CALL TSKVALUE(LCHILD)
C
C     Common code for all tasks.
C
   1  CONTINUE
C
  99  CONTINUE
      CALL LOCKON (LOCKI)
         IGLOBAL = IGLOBAL + 1
         I = IGLOBAL
      CALL LOCKOFF(LOCKI)
      IF ( I .GT. L) GOTO 1000
C
      DO 10 J = 1,N
         C(I,J) = 0
 10   CONTINUE
      DO 30 K = 1,M
         DO 20 J = 1,N
            C(I,J) = C(I,J) + A(I,K)*B(K,J)
  20     CONTINUE
  30  CONTINUE
      GOTO 99
1000  CONTINUE
C
C     The slave tasks return when their work is done; the master
C     task must wait until all slave tasks have completed.
C
      IF (LCHILD .NE. 0) RETURN
      DO 1001 I = 1,NPROC-1
         CALL TSKWAIT(TID(1,I))
1001  CONTINUE
      RETURN
      END

      SUBROUTINE TRANS2(A,L,B,M,C,N)
      CALL SELSYND2(A,L,B,M,C,N)
      RETURN
      END
```

10. MACROTASKING IN CAL

Multitasked programs may include user software written in languages other than Fortran. This section addresses some of the areas of special concern, focusing on CAL. Section 11, Multitasking with Pascal, covers Pascal multitasking in detail.

You can generally modify software written in CAL to work in a multitasked program, but you must be careful when doing so.

First, the subroutine should use the calling sequence first introduced with the COS 1.12 release. Using ENTER, EXIT, and associated macros is recommended. If the CAL subroutine is to be reentrant, it must use the stack calling sequence and should access local variables using the LOAD and STORE macros. CAL Version 2 supports a STACK SECTION option in which names can be assigned stack attributes. The size of such sections is passed to SEGLDR, and the assembler checks for improper use of these names in expressions. (The Macros and Opdefs Reference Manual describes these macros.) A subroutine coded with ENTER, EXIT, LOAD, STORE, and associated macros produces stack-based code if assembled with a version of the system text that has the stack flag set.

Second, you should ensure that global variables are stored in memory before subroutine calls that may lead to a task switch. COMMON blocks can be defined within CAL code for consistency with CFT77 or CFT code. CAL Version 2 supports a TASK COMMON SECTION option.

Third, you must be careful if I/O is included. The COS I/O tables (LFTs and DSPs), the datasets, and the I/O buffers may require treatment as shared variables requiring protection.

Finally, call the multitasking library subroutines as described in the Programmer's Library Reference Manual. Register preservation assumptions are the same as with any library call.

For examples of CAL subroutines that have been modified to work in a multitasking environment, you could look at subroutines in the various CRI default libraries.

10.1 PROCESSOR CLUSTERING

At the hardware level, multitasking involves grouping processors with a mechanism called *clustering*. Clustering allows several processors to communicate efficiently with one another and to coordinate and control their combined activity in executing a single multitasked job. The two-processor CRAY X-MP computer system has three clusters of shared registers, while the four-processor CRAY X-MP computer system has five clusters.

A processor is assigned to a cluster according to the Exchange Package that is active in the processor. The Exchange Package contains a field called cluster number (CLN) to which the operating system assigns a value. If each of several processors contains an Exchange Package with the same cluster number value, these processors are clustered together. Figure 10-1 shows the configuration for a four-processor CRAY X-MP computer system.

Any combination of processors can be clustered together. In a system with no multitasked jobs, each processor is assigned a different cluster, and independent jobs do not share a cluster. For a multitasked job, all the Exchange Packages of a job are assigned the same cluster number. Tasks executing on different processors cause the processors to be clustered together, which lets the tasks share the registers in the cluster.

10.1.1 SHARED REGISTERS

Processors clustered together may access a shared register set. Each cluster is identical and contains three groups of registers. Eight 24-bit shared address (SB) resisters, eight 64-bit shared scalar (ST) registers, and thirty-two 1-bit semaphore (SM) registers make up each cluster. The shared registers can be used to pass address and scalar information from one processor to another or for control among processors. Figure 10-2 shows shared registers in a cluster.

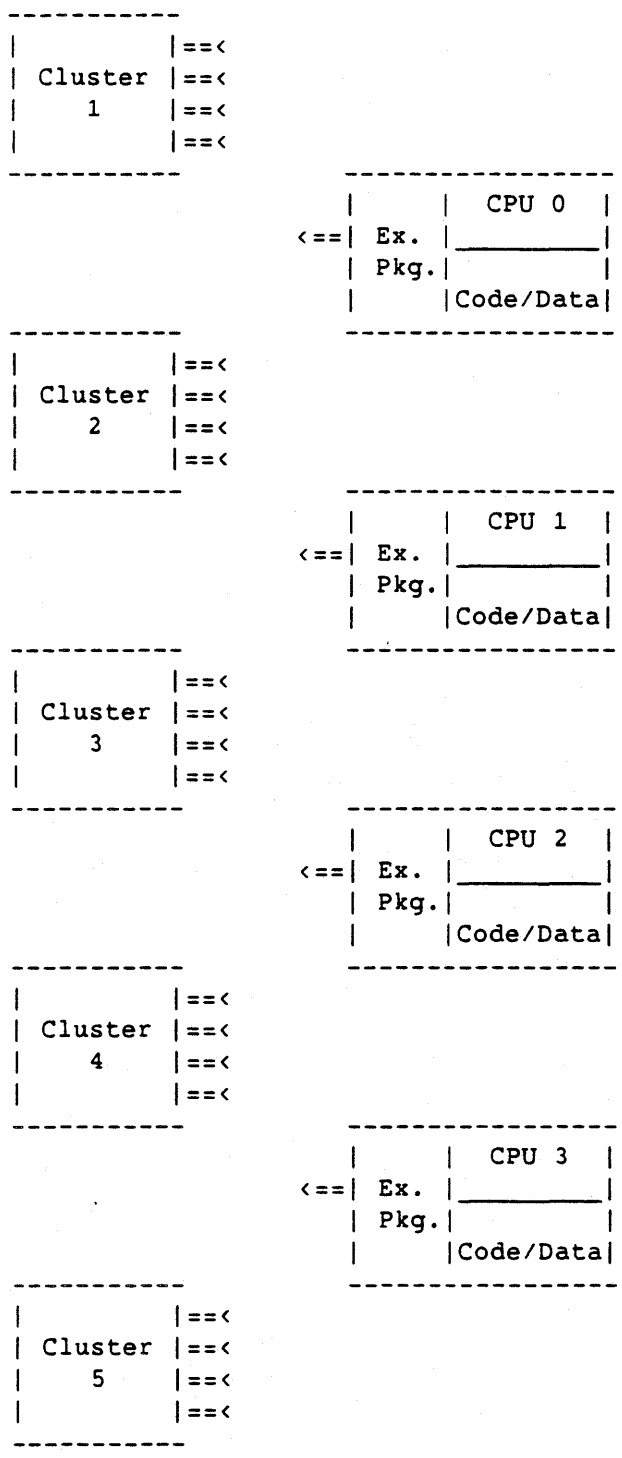


Figure 10-1. Clusters and Processors for a Four-processor CRAY X-MP Computer System

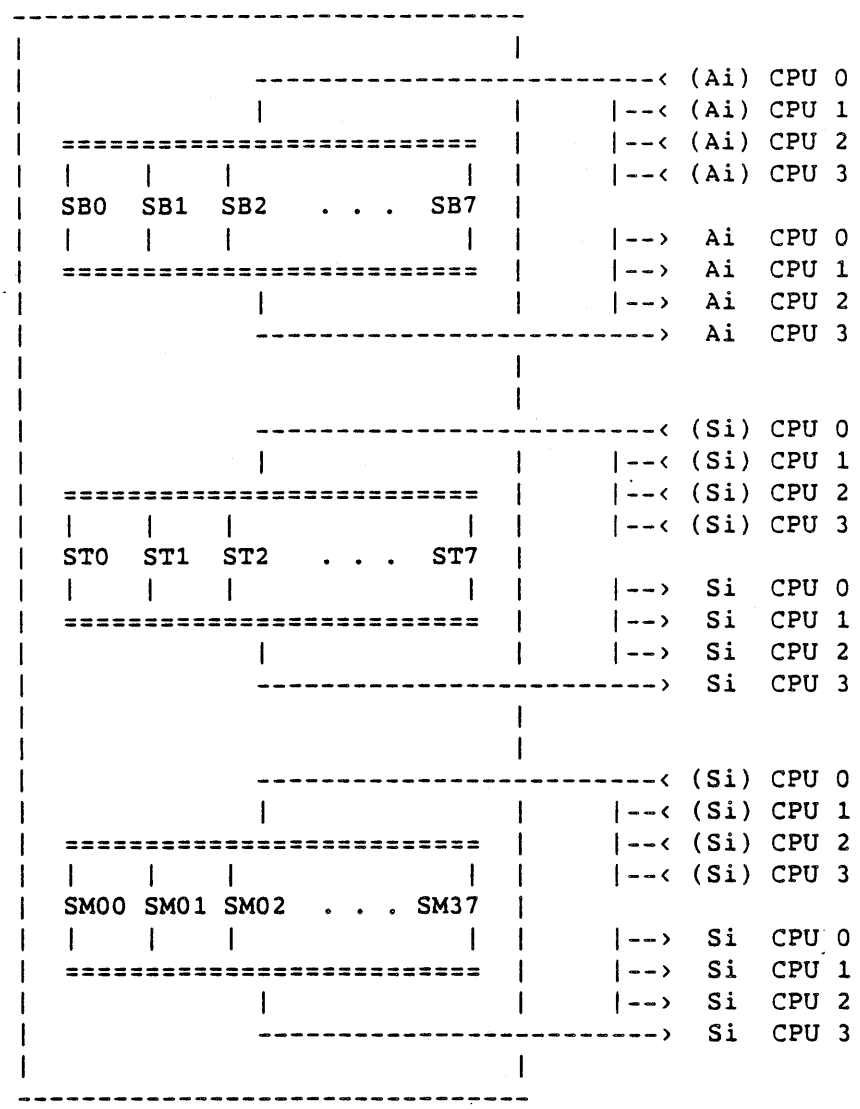


Figure 10-2. Shared Registers in a Cluster

NOTE

CAL subroutines used in a multitasking system must not use hardware semaphore registers 0 through 16; these registers are reserved for system use. Direct use of cluster registers and shared B and T registers should be done with care, because system software and CFT77 or CFT-generated code may use these currently or in the future.

10.1.2 MACHINE INSTRUCTIONS

The machine instructions that access the shared registers in a cluster are listed below. These instructions access the cluster to which the processor is assigned. You should consult the appropriate hardware mainframe reference manual for timing, reservation, and conflict resolution information.

Monitor mode:

0014j3	CLN	j	Enter j into CLN field of Exchange Package

User mode:

026ij7	Ai	SBj	Transmit (SBj) to Ai
027ij7	SBj	Ai	Transmit (Ai) to SBj
072ij3	Si	STj	Transmit (STj) to Si
073ij3	STj	Si	Transmit (Si) to STj
072i02	Si	SM	Transmit (SM) to Si
073i02	SM	Si	Transmit (Si) to SM
0034jk	SMjk	1,TS	Test and set SMjk
0036jk	SMjk	0	Clear SMjk
0037jk	SMjk	1	Set SMjk

11. MACROTASKING WITH PASCAL

Macrotasking can be accomplished within a Cray Pascal program in much the same way as within a Fortran program. Microtasking, however, is not possible with Pascal.

Only Pascal programs compiled with Pascal version 3.0 or later can be used with the multitasking library routines. Earlier versions of the Pascal compiler used a different stack management system that did not include tables used by the multitasking routines.

Pascal code used in a multitasked program must be compiled with the reentrant (Z+) option, which is turned on by default. The multitasking versions of all libraries with routines that will be used must be local to the job when the program is loaded, just as with multitasked Fortran programs. You can specify heap and stack initial sizes and increments at load time.

The multitasking library routines are accessed from Pascal in the same way that they are accessed from Fortran code. All of the routines must be declared as external procedures before they are used, and the $PSCLIB procedure P$TASK should be used in place of TSKSTART.

11.1 MULTITASKING PROCEDURES

P$TASK initiates a task because TSKSTART cannot be called directly from Pascal code due to the way in which procedure parameters are passed. If more than one procedure is used for starting new tasks, there must be a different version of the procedure declaration of P$TASK for each starting procedure, unless their argument list declarations are identical.

P$TASK and the multitasking library routines are not predefined Pascal procedures, so they must be declared in any Pascal modules that use them. Suggested declarations for these routines are given in subsection 11.9, Declarations for Multitasking Data Types and Procedures. Declarations equivalent to the following are needed to declare P$TASK:

```
TYPE
   TASKVALTYPE = one-word-data-type;
   TASKCONTROL = RECORD
                   TCSIZE: INTEGER;
                   TASKID: INTEGER;
                   TASKVAL: TASKVALTYPE;
                 END;
PROCEDURE taskproc [(list)];
. . . (body of procedure)
PROCEDURE TSKSTART (VAR TCS: TASKCONTROL;
                    PROCEDURE TASKPROC [(list);
                    list]);
   IMPORTED (P$TASK);
     . . .
TSKSTART(taskrecord,taskproc[,argumentlist]);
```

taskrecord Task control structure used for this task. Word 1 must be set; word 3, if used, must also be set. On return, word 2 is set to a unique task identifier that must not be changed later in the program because the multitasking library routines use this word. (See subsection 11.2, Task Control Structure.)

taskproc Procedure in which task execution begins. This procedure must be declared before it is used as an argument to P$TASK. It can be used as an argument only in procedures from which it could be called directly.

argumentlist List of arguments being passed to the procedure in which the new task begins execution. The types of items in this list must match the declared types of parameters in the procedure declaration for taskproc. All arguments passed to the new task must be passed as VAR parameters.

The following example shows two declarations for P$TASK. The two different declarations are necessary because the procedures in which new tasks begin have different argument lists.

Example:

```
PROGRAM multi;
CONST
  taskcount = 4;
TYPE
  taskvaltype = ALFA;
  taskcontrol = RECORD
                  tcsize: INTEGER;
                  taskid:  INTEGER;
                  taskval: taskvaltype;
                END;
  datalist = ARRAY [1..1000] OF REAL;
VAR
  temp1, temp2 : integer;
  tcs:  ARRAY [1..taskcount] OF taskcontrol;
  m:    ARRAY [1..taskcount] OF datalist;
  tcs2: taskcontrol;

PROCEDURE tskstart1(VAR tcs: taskcontrol;
                    PROCEDURE taskproc (VAR x: datalist);
                    VAR x: datalist); IMPORTED (P$TASK);
PROCEDURE tskstart2(VAR tcs: taskcontrol;
                    PROCEDURE taskproc (VAR i,j: integer);
                    VAR i,j: integer); IMPORTED (P$TASK);
PROCEDURE tskwait (VAR tcs: taskcontrol); EXTERNAL;

PROCEDURE exttask (VAR i,j: integer); EXTERNAL;

PROCEDURE task (VAR x: datalist);
(* declarations and body for procedure task *)

BEGIN

  (* Load data from some outside source *)
  ...

  (* Create tasks to each work on a portion of the data *)

  FOR i := 1 TO taskcount DO
    BEGIN
      WITH tcs[i] DO
        BEGIN
          tcsize := 3;
          taskname := 'TASK    ';
          taskname[6] := chr(ord('0')+i);
        END;
      tskstart1(tcs[i],task,m[i]);
    END;
```

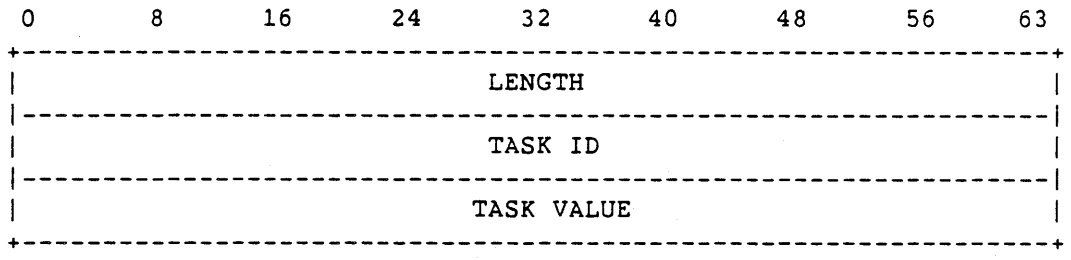
```
    (* Start an additional task in an external procedure *)

    tcs2.tcsize := 3;
    tcs2.taskname := 'EXT TASK';
    temp1 := 500;
    temp2 := 1000;
    tskstart2(tcs2,exttask,temp1,temp2);

    (* Wait for all tasks to finish *)

    FOR i := 1 TO taskcount DO
      tskwait(tcs[i]);
    tskwait(tcs2);

    (* Perform some post-execution cleanup *)
    ...

  END.
```

Functions LOCKTEST, EVTEST, and TSKTEST normally return Fortran logical results. You should declare them to return integer results (nonzero for true and zero for false) or replace them with local procedures that substitute Pascal Boolean results for Fortran logical results.

Unlike Fortran code, TSKTUNE in Pascal is declared with a fixed number of parameters and can be called only with that number of inputs. It is therefore suggested that TSKTUNE be declared with one keyword/value pair of parameters and simply called multiple times for multiple inputs (see the examples provided in subsection 11.9, Declarations for Multitasking Data Types and Procedures).

11.2 TASK CONTROL STRUCTURE

Each user-created task is represented by a task control structure, constructed by the user program. At a minimum, the structure must consist of 2 Cray words. A third word can be included. This structure can be either an integer array of 2 or 3 words or a record in which at least the first word is an integer. The structure of this array or record is as follows:



Field	Description

LENGTH Length of the structure in Cray words. The length must be set to a value of 2 or 3, depending on the optional presence of the task value field. You set the length field before creating the task.

TASK ID A task identifier assigned by the multitasking library when a task is created. This identifier is unique among active tasks within the job step. The multitasking library uses this field for task identification, but the task identifier is of limited use to user programs.

TASK VALUE (optional field)
Field that you can set to any value that fits in a Cray word before creating the task. If TASK VALUE is used, LENGTH must be set to a value of 3. The task value can be used for any purpose. Suggested values include a programmer-generated task name or identifier or a pointer to a task local storage area. During execution, a task can retrieve this value with the TSKVALUE procedure.

11.3 STORAGE OF TASK CONTROL STRUCTURES AND LOCK AND EVENT VARIABLES

Task control structures, lock variables, and event variables must be accessible to all procedures that use them. The actual storage location of a lock or event variable is manipulated by the multitasking library routines; therefore, all of these variables should be in static storage areas unless the stack frame for the procedure in which they are declared is guaranteed to exist for the entire time when the variables can be referenced. Variables declared in the main program or in the outermost level of a module are in static storage areas, as are variables declared in the STATIC declaration section of a nested procedure. Task control structures, lock variables, and event variables must be passed as VAR parameters.

11.4 ARGUMENTS PASSED TO A PROCEDURE HAVING A NEW TASK

Arguments passed to a procedure in which a new task begins must be passed as VAR parameters. If they are passed by value, the temporary storage in which their values reside can be reused any time after P$TASK returns to the calling task, even though the started task may not have executed.

VAR parameters to the new task are passed by address, becoming shared data whose subsequent use by different tasks must be synchronized.

As in Fortran programs that use multitasking, it is important to protect the use of shared variables by establishing lock variables for particular shared variables. These locks are then set and released around critical regions of code that use the shared variables.

11.5 PASSING PROCEDURAL AND FUNCTIONAL PARAMETERS TO A TASK

The Pascal compiler uses temporary storage areas for descriptors for procedural and functional parameters. If you pass procedures or functions to a new task, you must ensure that the parent task that called *task start* is suspended until the new task begins executing and copies the descriptors into its own storage locations. If you do not guarantee this situation, the parent task can reuse its temporary storage, overwriting the descriptors that the new task needs. To prevent this overwriting, call EVWAIT immediately after the call to P$TASK and call EVPOST within the procedure in which the new task begins.

Example:

```
VAR  task_started : INTEGER;

PROCEDURE task (procedure passed_proc);
BEGIN
  evpost (task_started);
  ...
  passed_proc;
END;

BEGIN
  ...
  evasgn (task_started);
  tskstart (tca, task, passed);
  evwait (task_started);
  ...
END;
```

11.6 USE OF NONLOCAL VARIABLES

A Pascal procedure C, nested within procedures B and A, can reference the variables declared in those two procedures. In a multitasked Pascal program, nonlocal variables can be referenced from different tasks. If B has variables that are referenced by C in a separate task, B must not return from its call until its variables are no longer needed by C in its task. You can ensure that B does not return by calling TSKWAIT from B, which called TSKSTART on C. Calling TSKWAIT is necessary because the variables declared in B are stored in a stack frame that is released when the call of B is completed. The following example shows this technique:

Example:

```
PROCEDURE A;
VAR x : REAL;    (* variable to be used in a separate task *)
  ...
  PROCEDURE B;
  VAR tca : taskcontrol;
    ...
    PROCEDURE C;     (* procedure to be executed as a separate task *)
    VAR y : REAL;
    BEGIN   (* procedure C *)
      ...
      y := x;   (* use of a nonlocal variable (x) from another task *)
      ...
    END;     (* procedure C *)
  BEGIN   (* procedure B *)
    ...
    tskstart (tca, C);
    ...
  END;     (* procedure B *)
BEGIN   (* procedure A *)
  ...
  x := 4.5;
  ...
  B;
  ...
END;     (* procedure A *)
```

In the previous example, there is no call to TSKWAIT for the task that executes procedure C. Procedure A could therefore release its stack space and return from its call before the new task (involving procedure C) even began execution. This would result in an invalid or undefined value for variable x in procedure C. To prevent this, TSKWAIT should be called from procedure B or the declaration of *tca* should be moved to procedure A and TSKWAIT then called from procedure A.

Variables declared in the main program or at the module level and referenced by nested procedures in separate tasks should not be assigned to B or T registers. If program- or module-level variables are used in separate tasks, add O=BREG=0:TREG=0 to the Pascal control statement. If any nonlocal variables are used in separate tasks, add O=BREG-:TREG- to the Pascal control statement.

11.7 INPUT AND OUTPUT IN MULTITASKED PASCAL

File buffer variables should be treated as shared data when file variables are used by more than one task.

Only one task at a time should attempt to read from a particular Pascal input file. Sections of code that use the standard Pascal input routines (RESET, GET, READ, and READLN) or that reference the file's buffer variable directly should be locked if the file could be read from more than one task at a time.

There are two methods of protecting an output dataset that will be used by more than one task at a time. The first method is to lock all sections of code that use the standard Pascal output routines (REWRITE, PUT, WRITE, and WRITELN) or change the file's buffer variable. The second method is to provide a local file variable for each task that will write to the dataset, and to connect each local file variable to the external dataset. Use the $PSCLIB procedure P$LSTREW should be used in place of REWRITE to avoid rewinding the dataset for each task.

The second method protects output to individual records, but it allows records from more than one task to be interspersed. If any task writes messages of more than one record that should be kept together, you should use the first method of protecting output files.

11.8 TASK COMMON IN PASCAL

Cray Fortran TASK COMMON blocks can be used in Pascal code as TASKVAR variables. These variables are treated the same as regular common blocks when O=Z- is used on the Pascal control statement and are the same as TASK COMMON when O=Z+ (the default) is used.

11.9 DECLARATIONS FOR MULTITASKING DATA TYPES AND PROCEDURES

You must declare all multitasking library routines before they can be used in a Pascal procedure. The following suggested declarations can be used to declare these procedures:

```
TYPE
  lock = INTEGER;
  event = INTEGER;
  taskvaltype = ALFA;(* or anything else that will fit in one word *)
  taskcontrol = RECORD
                  tcsize: INTEGER;
                  taskid: INTEGER;
                  tskval: taskvaltype; (* this one is optional *)
                END;

PROCEDURE tskwait  (tcs: taskcontrol); EXTERNAL;

PROCEDURE tskvalue (VAR tskval: taskvaltype); EXTERNAL;

FUNCTION  tsktest  (tcs: taskcontrol): INTEGER; EXTERNAL;
   or
FUNCTION  tsktest  (tcs: taskcontrol): BOOLEAN;
  FUNCTION fortran_tsktest (tcs: taskcontrol): INTEGER;
    IMPORTED (TSKTEST);
  BEGIN
    tsktest := fortran_tsktest(tcs) <> 0;
  END;

PROCEDURE tsktune  (keyword: ALFA; val: INTEGER); EXTERNAL;

PROCEDURE tsklist; EXTERNAL;

PROCEDURE lockasgn (VAR name: lock); EXTERNAL;
   or
PROCEDURE lockasgn (VAR name: lock; initval: INTEGER); EXTERNAL;

PROCEDURE lockon   (VAR name: lock); EXTERNAL;

PROCEDURE lockoff  (VAR name: lock); EXTERNAL;

PROCEDURE lockrel  (VAR name: lock); EXTERNAL;

FUNCTION  locktest (name: lock): INTEGER; EXTERNAL;
   or
FUNCTION  locktest (name: lock): BOOLEAN;
  FUNCTION fortran_locktest (name: lock): INTEGER;
    IMPORTED (LOCKTEST);
  BEGIN
    locktest := fortran_locktest(name) <> 0;
  END;
```

```
PROCEDURE evasgn   (VAR name: event); EXTERNAL;
   or
PROCEDURE evasgn   (VAR name: event; initval: INTEGER); EXTERNAL;

PROCEDURE evwait   (VAR name: event); EXTERNAL;

PROCEDURE evpost   (VAR name: event); EXTERNAL;

PROCEDURE evclear  (VAR name: event); EXTERNAL;

PROCEDURE evrel    (VAR name: event); EXTERNAL;

FUNCTION  evtest   (name: event): INTEGER; EXTERNAL;
   or
FUNCTION  evtest   (name: event): BOOLEAN;
  FUNCTION fortran_evtest (name: lock): INTEGER;
    IMPORTED (EVTEST);
  BEGIN
    evtest := fortran_evtest(name) <> 0;
  END;
```

APPENDIX SECTION

A. MULTITASKING ON A SINGLE-PROCESSOR CRAY X-MP COMPUTER SYSTEM

Any multitasked program that runs correctly on a single-processor CRAY X-MP computer system will run correctly on a multiple-processor CRAY X-MP computer system, and vice versa. All CRAY X-MP computer systems have hardware semaphores and operating system support for multiple logical CPUs.

A program whose source has been modified for multitasking should be recompiled and the absolute module rebuilt if transferred between machine types. Binaries, especially absolute binaries, are not transportable.

Multitasked codes that run correctly on a CRAY-1 computer system execute correctly on a CRAY X-MP computer system, but the converse is not always true. For example, a program could be set up in which the synchronization between two tasks is by way of COMMON variables in memory, and one task loops until a second task has updated the variables. Although this is not a recommended design, it could execute correctly on a CRAY X-MP computer system (if the machine is dedicated and the update period is long) because the operating system time slices the logical CPUs assigned to the user job. Over time, both tasks will execute.

Under the simulation mode, an opportunity may never arise for the library scheduler to swap control between the two tasks, because no explicit synchronization is performed. Hence, one task or the other could retain control, either looping or updating until the job's time limit is exceeded.

The COS system calls that create and delete tasks are available for CRAY-1 computer systems, but the lack of an intertask synchronization mechanism (as provided by CRAY X-MP computer system hardware) makes these generally useless.

B. MESSAGES

The following messages may be encountered during the development and testing of a multitasked application. For information on other messages, see the COS Message Manual, publication SR-0039.

AB199 - MAXIMUM USER TASKS PER JOB EXCEEDED
The user has created more than I@MAXNUT (a COS installation parameter) active logical CPUs. This occurs only if TSKTUNE was called with MAXCPU set above I@MAXNUT.

MT001 - FILE *filename* IS EMPTY
The file named on MTDUMP, DN = *filename* is empty. Use a file containing the unformatted contents of the multitasking history trace buffer.

MT002 - UNRECOGNIZED FORMAT ON MTDUMP
The parameter FORMAT = $ on MTDUMP accepts TOTALS, CHRON, SYNC, CPU, or STATUS as valid parameters; others are unrecognized. Use one of the correct formats.

UT013 - FATAL STACK OVERFLOW
Insufficient space is available for expansion of the stack. Insufficient space was allocated for the stack, and the increment is zero on the LDR STK or the SEGLDR STACK directive is zero.

UT015 - EVREL CALLED WITH TASKS WAITING FOR EVENT
An event variable was released with EVREL, but some task was waiting for it.

UT016 - LOCKREL CALLED WITH LOCK SET
A lock variable was released with LOCKREL but was currently in use by some task.

UT017 - INVALID LOCK IDENTIFIER
LOCKREL was called, but the specified lock variable appears to be invalid. Check to ensure that the lock variable was assigned and that it was not accidentally overwritten.

UT019 - HEAP IS FULL, CAN'T SATISFY REQUEST
Insufficient space is available for expansion of the heap. Either insufficient memory space remains in the job's field length or the increment on the LDR MM or SEGLDR HEAP directive is zero.

UT024 - DEADLOCK - ALL USER TASKS WAITING FOR LOCKS, EVENTS, OR TASKS
The library detected a situation in which all active tasks are suspended for events, locks, or other tasks.

UT025 - UNRECOGNIZED SCHEDULER PARAMETER NAME
An ASCII string passed as a TSKTUNE parameter was not recognized.

C. APPROXIMATE TIMINGS

This appendix contains approximate timings for the multitasking library subroutines. These timings are subject to change and are provided only for planning purposes. The timings are grouped by the subroutine groupings used in section 5, Macrotasking.

Parallelism subroutines:

Subroutine and Beginning Conditions	Clock Periods
TSKSTART (first call in program)	1,500,000†
(later call, if logical CPU†† needed)	40,000
(later call, if logical CPU not needed)	2,500
TSKWAIT (task completed execution)	400+†††
(task exists)	2,500+¶
TSKVALUE	150

Protection subroutines:

Subroutine and Beginning Conditions	Clock Periods
LOCKASGN	400
LOCKON (lock free)	400
(lock locked)	1,500+¶¶
LOCKOFF (no tasks waiting)	400
(tasks waiting)	1,800
LOCKREL	400

† The value of 1,500,000 for TSKSTART is a worst case and may occur when a memory expansion must obtain stack space. Parameters on the SEGLDR control statement can bring about this memory allocation at load time rather than at run time (see section 5, Macrotasking). Experience has shown that for most multitasking applications, the initial one or two TSKSTART calls are subject to the larger times shown above, while all subsequent calls take only about 2500 clock periods. Section 5 describes the events that could lead to longer times, but these are typically uncommon.

†† Logical CPUs are discussed in subsection F.1.1.

††† Plus approximately 25 clock periods for each existing task

¶ Plus time spent waiting for task to complete execution

¶¶ Plus time spent waiting for lock to be unlocked

Synchronization subroutines:

Subroutine and Beginning Conditions	Clock Periods
EVASGN	400
EVWAIT (event posted)	300
(event clear)	1,800+†
EVPOST (no tasks waiting)	400
(tasks waiting)	1,900
EVCLEAR	300
EVREL	400

† Plus time spent waiting for event to be posted

D. MULTITASKING STATUS FEATURES

Three function subprograms (one each for tasks, locks, and events) can be used to obtain the status of an entity. The following descriptions are not described in section 5, Macrotasking, because the subprograms are of limited use and carry a high risk. The risk is that a task using one of these subroutines may unintentionally enter a busy wait or a spin lock condition, locking all other tasks out of execution. For this reason, using these features in general code is discouraged.

D.1 TSKTEST

TSKTEST returns a value indicating whether or not the indicated task exists. TSKTEST must be declared LOGICAL in the calling module.

Format:

```
 ____________________________
|                            |
| return=TSKTEST (taskarray) |
|____________________________|
```

return A logical .TRUE. if the indicated task exists; a logical .FALSE. if the task was never created or has completed execution.

taskarray Task control array

D.2 LOCKTEST

LOCKTEST tests to see whether a lock is in the locked state. LOCKTEST acts the same as LOCKON, except that the task never waits. A task using LOCKTEST must always look at the return value before continuing. LOCKTEST must be declared LOGICAL in the calling module.

Format:

```
 ____________________________
|                            |
| return=LOCKTEST (name)     |
|____________________________|
```

return A logical .TRUE. if the lock was originally in the locked state; a logical .FALSE. if the lock was originally in the unlocked state. The lock variable's state is always set to locked upon return.

name Name of an integer variable used as a lock

D.3 EVTEST

EVTEST tests whether an event is posted. It must be declared LOGICAL in the calling module.

Format:

```
 ____________________________
|                            |
| return=EVTEST (name)       |
|____________________________|
```

return A logical .TRUE. if the event is posted; a logical .FALSE. if the event has never been posted or is cleared. The event variable's state is unaffected by a call to EVTEST.

name Name of an integer variable used as an event

E. BIBLIOGRAPHY

The following references contain additional information on multitasking and its applications:

Andrews, G. and F. Schneider. "Concepts and Notations for Concurrent Programming." *ACM Computing Surveys*, vol. 15, no. 1 (March 1983): 3-43.

Baer, J. L. "A Survey of Some Theoretical Aspects of Multiprocessing." *ACM Computing Surveys*, vol. 5, no. 1 (March 1973): 31-80.

Ben-Ari, M. *Principles of Concurrent Programming*. London: Prentice-Hall International, 1982.

Bernstein, A. J. "Analysis of Programs for Parallel Processing." *IEEE Trans. Elec. Comp.*, vol. 15 (October 1966): 746-757.

Brinch Hansen, P. "Concurrent Programming Concepts." *ACM Computing Surveys*, vol. 5, no. 4 (December 1973): 223-245.

Bucher, I. Y. "The Computational Speed of Supercomputers." *Proceedings of ACM/SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (August 1983): 151-165.

Calahan, D. A. "Influence of Task Granularity on Vector Multiprocessor Performance." *Proceedings of the 1984 International Conference on Parallel Processing*. Ed. by Keller. IEEE Computer Society Press (August 1984): 278-284.

Chen, S. C., J. J. Dongarra, and C. C. Hsiung. "Multiprocessing Linear Algebra Algorithms on the CRAY X-MP-2: Experiences with Small Granularity." *Journal of Parallel and Distributed Computing*, vol. 1, no. 1 (August 1984): 22-31.

Cohen, T. "Structured Flowcharts for Multiprocessing." *Computer Languages*, vol. 13, no. 4 (1978): 209-226.

Dijkstra, E. W. "Co-operating Sequential Processes." *Programming Languages*. Ed. by E. Gunuys. New York: Academic Press (1968): 43-112.

Engel, T. M. and R. E. Wellck. "Multitasking of Atmospheric Forecast Models." Technical Report No. WSG-005. Boulder, CO: Cray Research, Inc., December 1983.

Enslow, P. H. "Multiprocessor Organization." *ACM Computing Surveys*, vol. 9, no. 1 (March 1977): 103-129.

_____, ed. *Multiprocessors and Parallel Processing*. New York: Wiley-Interscience, 1974.

Hansen, P. B. *The Architecture of Concurrent Programs*. Englewood Cliffs, NJ: Prentice-Hall, 1977.

Heller, D. "A Survey of Parallel Algorithms in Numerical Linear Algebra." *SIAM Review*, vol. 20 (1978): 740-777.

Hockney, R. W. and C. R. Jesshope. *Parallel Computers: Architecture, Programming and Algorithms*. Bristol, England: Adam Hilger Ltd., 1981.

Hsiung, C. C. and V. Butscher. "A New Numerical Seismic 3-D Migration Model for Vector Multiprocessors." *Parallel Computing*, vol. 1, no. 2 (1984).

Hwang, K. and F. A. Briggs. *Computer Architecture and Parallel Processing*. New York: McGraw-Hill, 1984.

Hwang, K. and R. H. Kuhn, eds. *Tutorial on Supercomputers Design and Applications*. IEEE Computer Society Press, 1984.

Kneis, W. "Industrial Real-Time Fortran Standard." *SIGPLAN Notices* (July 1981): 45-60.

Kuck, D. J. "Parallel Processing of Ordinary Programs." *Advances in Computers*, vol. 15. Ed. by Rubinoff and Yovits. New York: Academic Press (1976): 119-179.

_____ "A Survey of Parallel Machine Organization and Programming." *ACM Computing Surveys*, vol. 9, no. 1 (March 1977): 29-59.

_____ *The Structure of Computers and Computations*, vol. 1. New York: John Wiley and Sons, 1978.

_____ *High Speed Computer and Algorithm Organization*. Ed. by D. H. Lawrie and A. H. Sameh. New York: Academic Press, 1977.

Kuhn, R. H. *Tutorial on Parallel Processing*. Ed. by D. A. Padua. Los Angeles: IEEE Computer Society Press, order number 367, 1981.

Larson, J. L. "Multitasking on the CRAY X-MP-2 Multiprocessor." *IEEE Computer*, vol. 17, no. 7 (July 1984): 62-69.

Miranker, W. L. "A Survey of Parallelism in Numerical Analysis." *SIAM Review*, vol. 13 (1971): 524-547.

Padua, D. A., D. J. Kuck, and D. H. Lawrie. "High-Speed Multiprocessors and Compilation Techniques." *IEEE Trans. on Comp.*, vol. 29 (September 1980): 763-776.

Poole, W. G., Jr. and R. G. Voigt. "Numerical Algorithms for Parallel and Vector Computers: An Annotated Bibliography." *Computing Reviews*, vol. 15 (1974): 379-388.

Rodrique, G., ed. *Parallel Computations*. New York: Academic Press, 1982.

Sameh, A. H. "Numerical Parallel Algorithms--A Survey." *High-Speed Computers and Algorithm Organization*. Ed. by Kuck, et al. Academic Press, 1977: 207-228.

F. DESIGN DESCRIPTION

This section presents an overview of the design of the multitasking library subroutines and provides detail about the library subroutines.

F.1 LIBRARY SCHEDULER

The multitasking library is the primary manager and scheduler of tasks within a program. This approach offers advantages in the following areas:

- Performance; you can perform multiple operations at the library level without calls to the operating system.
- Tunability; you can tune the library for individual user programs, tuning it differently for different programs running simultaneously.
- Flexibility; making library changes is easier than making the same changes to the operating system.
- Ease of use; you are not required to maintain queues or task information, or to program in CAL in order to use the hardware multitasking features.

F.1.1 LOGICAL CPU

The logical CPU is the key concept of the operating system interface to the library scheduler. (The logical CPU is referred to in COS documentation as a *user task* and in UNICOS as a *process*.) A logical CPU is the entity that the operating system schedules for execution on physical CPUs, and it is identified as an entry in the COS Task Execution Table (TXT), or the UNICOS Process Table.

Initially, a job is assigned one logical CPU, but the library scheduler can request additional logical CPUs for a particular job, thereby bringing about multitasking. The number of logical CPUs need not, however, equal the number of tasks active in the user job. The maximum number of logical CPUs is a major tuning component of the library scheduler (see subsection 5.4, Tuning).

The job of the library scheduler, therefore, is to connect user tasks to logical CPUs in the most efficient manner. If a task must wait for a lock or an event, that task is disconnected from its logical CPU so that the logical CPU is freed for use by another task in the job or possibly for return to the system.

In many multitasking applications, the concept of the logical CPU may seem redundant or unnecessary. For example, if only two tasks are active on a CRAY X-MP computer system, the library scheduler would likely allocate and use two logical CPUs. The concept becomes important when there are more defined tasks than physical CPUs. Generally, the number of logical CPUs should not be greater than the number of physical CPUs to allow task scheduling to occur at the higher (and faster) level.

F.1.2 QUEUE MANAGEMENT

The library scheduler manages several queues of tasks. Tasks are moved between queues as their states change through the use of the multitasking facilities.

The queues are generally handled in first-in, first-out (FIFO) order. However, when a task calls any of the multitasking subroutines, it is placed in the front of the Waiting for Logical CPU queue. If a task is moved from one queue to another, it is placed at the end of the new queue. The following discussion describes only two queues: the Waiting for Logical CPU queue and the Suspended queue. (Multiple suspended queues are actually implemented.)

A number of the library subroutines exit through the library scheduler. The scheduler selects the first task in the Waiting for Logical CPU queue. If the task selected is the one that made the library call, that task is already connected to a logical CPU. The scheduler simply returns, and execution resumes in the user program. If a task other than the one that made the call is selected, registers for the task are loaded and execution resumes in the new task.

NOTE

B and T registers are explicitly loaded by the library scheduler; A, S, and V registers are loaded as needed within the user code. Because the multitasking features are implemented as library subroutines, programmers cannot assume that the contents of A, S, and V registers are preserved across the call or that the generated code or the CAL code performs its own reloading as required.

F.2 KEY LIBRARY SUBROUTINES

The following subsections describe the key library subroutines in detail.

F.2.1 TSKSTART

TSKSTART builds a stack for the task, copying initial information from the task control array into a Task Information Block at the base of the stack. The task is then placed in the Waiting for Logical CPU queue, and control passes to the library scheduler.

F.2.2 TSKWAIT

TSKWAIT checks the status of the specified task, returning control to the calling task if the specified task has completed execution. If the specified task is active, the calling task is placed in a Suspended queue, the identifier of the task for which it is waiting is saved, and control passes to the library scheduler.

F.2.3 LOCKON

LOCKON checks the status of the lock variable. If the lock variable is unlocked, the subroutine locks it and returns control. If the lock variable is locked, the calling task is placed in a Suspended queue, the identifier of the lock for which the task is waiting is saved, and control passes to the library scheduler.

F.2.4 LOCKOFF

LOCKOFF changes the status of the lock variable to unlocked. LOCKOFF then removes the first task waiting for that lock from a Suspended queue and puts it in the Waiting for Logical CPU queue. Control passes to the library scheduler.

F.2.5 EVWAIT

EVWAIT checks the status of the event. If the status is posted, control returns to the calling task without further action. If the status is cleared, the task is put in a Suspended queue, the identifier of the event for which it is waiting is saved, and control passes to the library scheduler.

F.2.6 EVPOST

EVPOST changes the status of the event variable to posted. EVPOST then removes all tasks waiting for that event from a Suspended queue and puts them in the Waiting for Logical CPU queue. Control passes to the library scheduler.

F.2.7 EVCLEAR

EVCLEAR changes the status of the event variable to cleared, and control returns to the calling task.

F.3 STATE TRANSITIONS

The multitasking routines and library scheduler, previously described, cause user tasks to move from state to state over the course of a job. Figure F-1 shows these transitions.

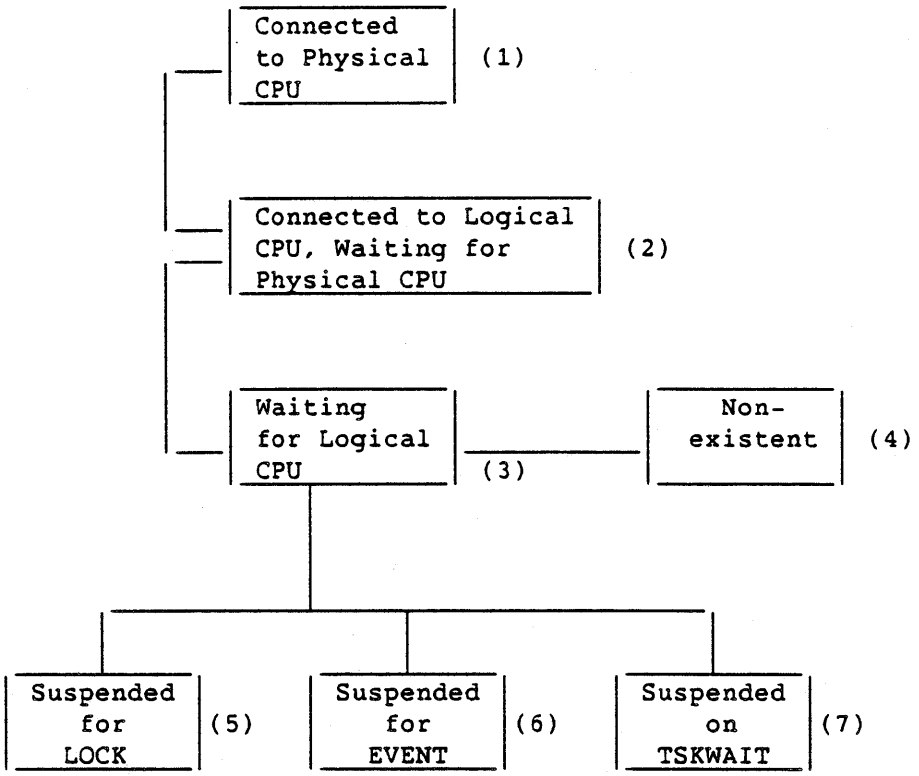


Figure F-1. Transitions of User Tasks

The descriptions of transitions between states in figure F-1 are as follows:

Transition	Description
(1)-->(2)	Because of an interrupt, the task is removed from a physical CPU.
(2)-->(1)	The operating system selects the task for execution on a physical CPU.
(2)-->(3)	A multitasked routine is called. (This transition may be transitory and may be immediately followed by the opposite transition.)
(3)-->(2)	The library scheduler assigns the task to a logical CPU, using the internal FIFO queue.
(3)-->(4)	The task completes execution.
(4)-->(3)	The task is created through a call to TSKSTART.
(3)-->(5)	The task executes LOCKON for a lock that is already on.
(5)-->(3)	Some other task executed LOCKOFF, and this task is selected to receive the lock.
(3)-->(6)	The task executes EVWAIT for an event that is not posted.
(6)-->(3)	Another task posts the event for which this task is waiting.
(3)-->(7)	The task executes TSKWAIT for an existing task.
(7)-->(3)	The task being waited for completes execution.

F.4 TASK COMMON

When you use static allocation, TASK COMMON is treated the same as COMMON. When you use stack allocation, SEGLDR and LDR set up a table with the externally accessible name of $TASKCOM, containing the following information:

- The number of TASK COMMON blocks
- The sum of their lengths

- For each TASK COMMON block, its ASCII name, its length, and its address

The loaders consider the address of a TASK COMMON block to be the offset into $TASKCOM of the address/offset field.

TSKSTART performs the following actions at run time to set up TASK COMMON blocks:

- Copies the table from the externally accessible prototype into the stack for the new task
- Allocates memory space for the TASK COMMON blocks and places the address for each block into the table
- Places the address of word 0 of the table into a field at the top of the stack

This implementation allocates space for all TASK COMMON blocks to each task, regardless of whether or not it uses them.

F.5 MEMORY MANAGEMENT

The following levels of memory management exist within a multitasked program:

- Heap storage
- Stacks
- Activation blocks

F.5.1 HEAP

The *heap* is an area of memory within the user field length managed by user-callable library routines that provides dynamic storage allocation for a single job. Initial heap space is allocated at load time; the heap manager library routines request additional memory from the operating system when they are unable to satisfy requests using the free space already on the heap.

The heap generally follows blank COMMON. It can be placed after the user code and data and before blank COMMON if the program does its own memory management. The heap is not allowed to expand when it precedes blank COMMON.

The initial size, increment size, and location (before or after blank COMMON) of the heap are specified at load time, as described in subsection 5.4.2, LDR and SEGLDR Memory Management Tunings.

F.5.2 STACKS

A *stack* is a linear list data structure for which all accesses are made from one end. Additions to the list are made by *pushing down* an item onto the top of the stack, and deletions involve *popping up* an item from the top of the stack. The last item entered on the stack is the first item out.

Stacks provide the reentrant property needed for tasks and subroutines in a multitasking environment. Reentrant Fortran and CAL routines use stack space managed by the stack manager library routines. Stacks are allocated from the heap. Each task has its own stack with one or more stack segments. The initial stack segment for a task is set up when the task is initialized and includes a task information block used by the multitasking library routines, a table with TASK COMMON block names and addresses, and the TASK COMMON blocks themselves. Each active subroutine in the task has an activation block in the stack.

Additional stack segments are added when a stack overflows. Each stack segment is linked to the previous segment and to the stack header at the base of the task's initial stack segment.

The stack mechanism provides for reentrancy by giving each task an independent local workspace in TASK COMMON blocks that can be shared by all subroutines in a task. Additionally, a single subroutine can be used by more than one task, because the necessary independence required for reentrancy is provided by a separate activation block for each occurrence of the subroutine.

The initial size and increment size of the stack are specified at load time, as described subsection 5.4.2, LDR and SEGLDR Memory Management Tunings.

A portion of the user's area in memory is allocated for stack space and is called *managed memory*. The default location for managed memory is following the blank COMMON area, as indicated in figure F-2.

Within the managed memory area are the stacks associated with tasks in the job. Each task has its own stack, and items in stacks are called *stack frames*. At the base of each stack is a task information block containing data such as the task ID, task value, and other information used by the multitasking library. The next frame contains any defined TASK COMMON blocks.

When a task calls a subroutine, a new frame (called an *activation block*) is pushed onto the stack. If this subroutine calls another subroutine, a new activation block is added on top of the previous one (see figure F-3). The number of activation blocks on the stack reflects the current depth level of subroutine calls. When a subroutine returns, its activation block is popped off the stack.

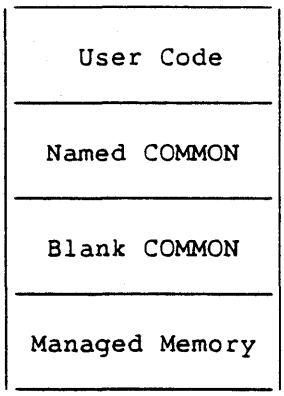


Figure F-2. User Area in Memory

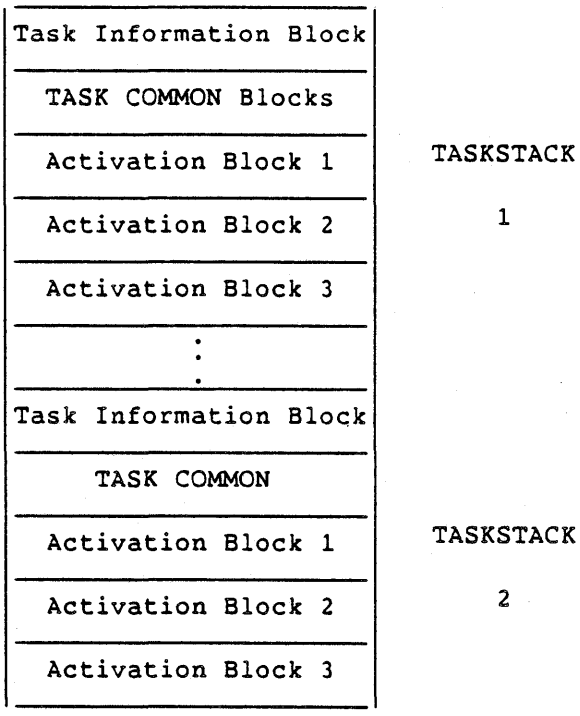


Figure F-3. Task Stacks in Managed Memory

F.5.3 ACTIVATION BLOCKS

Each active subroutine in a task has an activation block on the stack. This activation block contains local variables, save areas for B and T registers, and temporary storage. An activation block is pushed onto the stack in the subroutine entry processing and is popped off in the exit processing. The number of activation blocks on the stack reflects the current depth level of subroutine calls in the task. Figure F-4 illustrates an activation block.

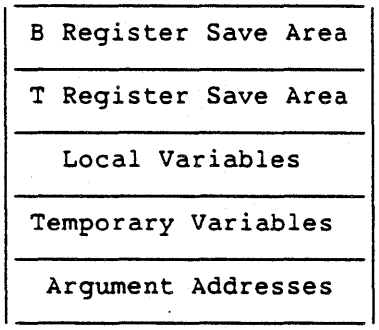


Figure F-4. Activation Block Stack Frame

Register B02 is the current base pointer, pointing to the base of the activation block for the subroutine that is executing. Register B66 is the current top pointer, pointing to the first word past the current activation block. Fortran code uses B03 to point to the beginning of the local variables for the subroutine that is executing.

During the entry sequence, stack pointers in registers B02 (the current base pointer) and B66 (the current top pointer) are updated to delimit the activation block for the routine just entered. An overflow occurs when the stack does not contain enough memory for the new routine, causing B66 to be greater than B67 (the absolute top pointer). The stack overflow library routine is then called to extend the stack segment or add a new stack segment.

During the exit sequence, the stack pointers for the calling routine are restored. When a routine that caused an overflow returns, the stack underflow library routine is called to release the additional stack segment. The stack space for a subroutine is reused after the subroutine returns.

Figure F-5 shows the division of memory in the user area. (The } symbols indicate areas that are expanded in other parts of the figure.)

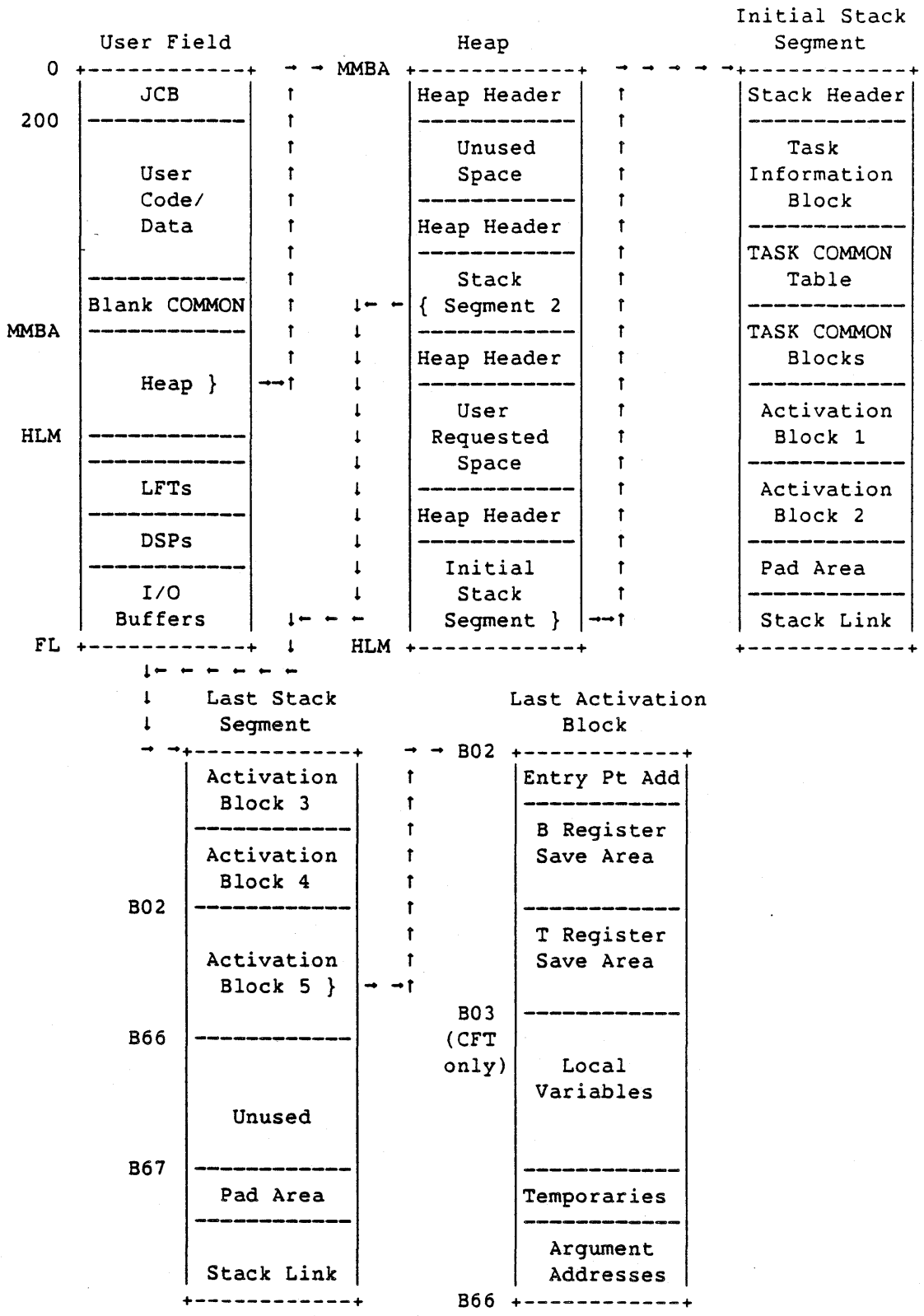


Figure F-5. Division of Memory in the User Area

G. NOTES ON MULTITASKING

This appendix contains miscellaneous notes on multitasking.

G.1 USING COS MULTITASKING MACROS

The Macros and Opdefs Reference Manual describes several macros that directly create and delete logical CPUs from user code. The library scheduler uses these macros and programmers can also use the macros, but they will be unable to synchronize between tasks created in this manner and any tasks created using the library routines described in this manual. In general, macros and library calls should be viewed as mutually exclusive.

G.2 BATCH USE OF MULTITASKING

As mentioned in section 1, Introduction, multitasking is aimed at the dedicated environment, though jobs can be run in a batch environment and batch processing can be useful for program development and debugging. Some suggestions for more effective batch use follow:

- Subsection 5.4, Tuning, includes a discussion of the tuning subroutine, TSKTUNE. One of the parameters to this routine defines the maximum number of CPUs that will execute in an idle loop if otherwise unneeded. If a program tends to keep CPUs unused for long periods of time, you should consider setting the parameter to zero in a production batch environment. Otherwise, the CPU will execute the idle loop and be unavailable to other jobs in the system.

- Performance in a batch environment is highly variable. If performance testing during batch is considered important, a site could establish a job class for multitasking that assigns a higher priority to such jobs. However, tasks in such jobs are still scheduled individually, depending upon their priority and those of other jobs present in the system. Sites should not introduce such a job class without carefully considering the impact on system throughput.

- Using microtasking in a batch environment is strongly encouraged. A microtasked application will use more than one CPU only if those CPUs would otherwise be idle. If no extra CPUs are ever available, the application should only minimally increase its execution time. Using microtasking, especially in situations when CPUs are frequently idle, improves both job and system throughput.

GLOSSARY

L

Library scheduler - A library subroutine that assumes primary responsibility for managing and scheduling the tasks within a program

Library task - A task created by a user job using the library calls described in this manual. A library task is referred to simply as a task in this manual.

Load balancing - A process used to ensure that the amount of work done by each of the processors involved in a job is approximately equal

Local - Accessible only to a particular part of a program (usually a single module). Local is a type of scope.

Lock - (1) A facility that monitors critical regions of code. Locks have two states: locked and unlocked. (2) A variable used to represent a lock.

Lock variable - An integer variable representing a lock. This term is synonymous with *lock*.

Logical CPU - An entity scheduled by COS for execution on physical CPUs: a user task.

M

Macrotasking - An implementation of multitasking that allows parallel execution of code at the subroutine level on multiple processors.

Managed memory - A portion of the user's area in memory that is allocated for stack space and has a default location following the blank COMMON area.

Microtasking - An implementation of multitasking that allows parallel execution of very small segments of code, such as individual iterations of DO loops, on multiple processors.

Monitor - Controlling access to critical regions

Multiprocessing - A property of the hardware in which two or more CPUs are available

Multiprogramming - A property of the operating system that permits overlapping and interleaving the execution of more than one program

Multitasking - The structuring of a program into two or more tasks that can execute concurrently on two or more processors

Multithreading - See *reentrancy*.

Mutual exclusion - A property of a critical region in which no more than one task can execute it at a time

N

Nondeterministic - Not able to determine from the start. Multitasking is nondeterministic with respect to time; the order of execution of parallel tasks cannot be determined from run to run.

Nonreentrancy - A property of a program module that allows it to be used only once. Such a module is called nonreentrant.

P

Parallelism - The simultaneous processing of jobs, parts of jobs, programs, or parts of programs. The order of execution for code segments that execute in parallel typically cannot be determined ahead of time.

Parallel processing - The simultaneous processing of two or more segments of code

Post - An operation causing the event state to change to posted

Posted - An event state indicating that a signal is outstanding

Private - Local to a task: existing only within the task's scope boundaries. Private is a type of scope.

Process - In microtasking, a portion of work that will be completed by only one processor

R

Reentrancy - A property of a program module that allows one copy of it to be used by more than one job or task in parallel. Such a module is called reentrant.

Release - Indicating that a variable is no longer intended for use as a lock or event

S

Scope - The region of a program in which a variable is defined and can be referenced. See local, common, and shared.

Scope boundaries - The beginning and end of the region of a program that is the scope of a variable

Serial reusability - A property of a program module that allows it to be used multiple times but by no more than one task at a time

Shared - Accessible by multiple parts of a program. Shared is a type of scope.

Single threading - Limiting the executing of a region of code to a single task

Spin off - The process by which a task calls subroutines that are then made into independent tasks and run in parallel with the original

Stack - A data structure providing a dynamic, sequential data list having special provisions for access from one end or the other. A last-in, first-out (push down, pop up) stack is accessed from just one end.

Stackframe - An element of a stack. A stackframe is allocated when a reentrant subroutine is entered and deallocated on exit.

Starvation - A characteristic of a multitasking program in which one or more tasks get no (or virtually no) execution time on a physical CPU

Storage dependence - A form of dependence that occurs when tasks share variables

Synchronization - The process of coordinating the steps within processes that can be run in parallel

Synchronization point - A point in time at which a task has received the go-ahead to proceed with its processing

System task - One of the modules that constitute the Cray operating system COS; for example, Disk Queue Manager (DQM) or Station Call Processor (SCP).

T

Task - A software process. A task is a unit of computation that can be scheduled and whose instructions must be processed in sequential order; a subprogram.

TASK COMMON - Data that must be common to all subroutines that are executed by a single task but should be local to that task

Task control array - A data structure used to represent a user-created task

Task granularity - The approximate execution time of a task, usually given as an order of magnitude

Task information block - An area in the base of the task's stack that contains internal information about the task

Task value - An optional word within a task control array that may be set to any value by the user before creating the task

U

User task - An entity scheduled for execution by COS. A user task is referred to as a logical CPU in this manual.

INDEX

INDEX

READER'S COMMENT FORM

CRAY X-MP Multitasking Programmer's Manual SR-0222 I

Your reactions to this manual will help us provide you with better documentation. Please take a moment to check the spaces below, and use the blank space for additional comments.

1) Your experience with computers: ____ 0-1 year ____1-5 years ____5+ years
2) Your experience with Cray computer systems: ____0-1 year ____ 1-5 years ____5+ years
3) Your occupation: ____ computer programmer ____ non-computer professional
____ other (please specify): _____
4) How you used this manual: ____ in a class ____as a tutorial or introduction ____ as a reference guide
____ for troubleshooting

Using a scale from 1 (poor) to 10 (excellent), please rate this manual on the following criteria:

5) Accuracy _____
6) Completeness ____
7) Organization ____
8) Physical qualities (binding, printing) ____
9) Readability ____
10) Amount and quality of examples ____

Please use the space below, and an additional sheet if necessary, for your other comments about this manual. If you have discovered any inaccuracies or omissions, please give us the page number on which the problem occurred. We promise a quick reply to your comments and questions.

Name _____
Title _____
Company _____
Telephone _____
Today's Date _____

Address _____
City _____
State/ Country _____
Zip Code _____

CUT ALONG THIS LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

Attention: PUBLICATIONS
1345 Northland Drive
Mendota Heights, MN 55120

STAPLE

READER'S COMMENT FORM

CRAY X-MP Multitasking Programmer's Manual SR-0222 I

Your reactions to this manual will help us provide you with better documentation. Please take a moment t check the spaces below, and use the blank space for additional comments.

1) Your experience with computers: ____ 0-1 year ____1-5 years ____5+ years
2) Your experience with Cray computer systems: ____0-1 year ____ 1-5 years ____5+ years
3) Your occupation: ____ computer programmer ____ non-computer professional
____ other (please specify): _____
4) How you used this manual: ____ in a class ____as a tutorial or introduction ____ as a reference guide
____for troubleshooting

Using a scale from 1 (poor) to 10 (excellent), please rate this manual on the following criteria:

5) Accuracy _____
6) Completeness ____
7) Organization ____
8) Physical qualities (binding, printing) ____
9) Readability ____
10) Amount and quality of examples ____

Please use the space below, and an additional sheet if necessary, for your other comments about this manual. If you have discovered any inaccuracies or omissions, please give us the page number on which the problem occurred. We promise a quick reply to your comments and questions.

Name _____
Title _____
Company _____
Telephone _____
Today's Date _____

Address _____
City _____
State/ Country _____
Zip Code _____