MTASK

Parallel Programming Language

Version 2.1

List of Contents

List of Contents …… i
About this Manual …… ii
Assumption About the Reader …… ii
Overview of This Manual …… ii

Chapter 1. Introduction …… 1
1.1 Introduction …… 1
1.2 Manipulation of Tasks …… 1
1.3 Shared and Private Data …… 1
1.4 Data Dependencies …… 2
1.5 Task Synchronization …… 3
1.6 Function Partitioning …… 4
1.7 Data Partitioning …… 4
1.8 Data Distribution …… 4
1.9 Types of Parallelism …… 5
1.10 Lifetime of Variables …… 5

Chapter 2. Manipulation of Tasks …… 7
2.1 Creation of Tasks …… 7
2.2 Waiting For the Completion of Tasks …… 10
2.3 Termination of Tasks …… 11
2.4 Waiting For and Terminating Tasks …… 12

Chapter 3. Manipulation of Parallel Locks …… 13
3.1 Definitions …… 13
3.2 Critical Section …… 14

Chapter 4. Manipulation of Parallel Events …… 15
4.1 A Cycle of an Event …… 15
4.2 Definitions …… 15

About This Manual

This manual introduces basic concepts of parallel programming, and explains how to use MTASK to develop parallel programs. MTASK has subroutines for manipulations of task, parallel locks, and parallel events. MTASK not only works on multiprocessor computers but also works on uniprocessor computer. This manual has essential concept and practical techniques for programming in MTASK.

Assumptions About the Reader

This manual assumes that reader has experience writing, executing, and debugging Fortran program.

Overview of This Manual

This manual is organized as follows:

Chapter 1	**Introduction**. This chapter introduces terms and concepts that the user will need to be familiar with before programming a multitasking application.
Chapter 2	**Manipulation of Tasks**. This chapter describes subroutines for manipulation of tasks, calling syntax, and where in the program the subroutines are called.
Chapter 3	**Manipulation of Parallel Locks**. This chapter describes subroutines for manipulation of parallel locks, calling syntax, and where in the program the subroutines are called.
Chapter 4	**Manipulation of Parallel Events**. This chapter describes subroutines for manipulation of parallel events, calling syntax, and where in the program the subroutines are called.

Chapter 1. Introduction

1.1 Introduction

With achievements of computer hardware and new generation of operating systems, high performance computing becomes available in many fields of application. This manual introduces MTASK for developing parallel programs. MTASK based upon multitasking is a tool for developing parallel applications. Multitasking is a programming technique that allows a single application to consist of multiple tasks executing concurrently, and yields improved execution speed for individual programs if multiprocessors are available.

1.2 Manipulation of Tasks

A task is analogous to a subroutine (or a function) in a program, except for the important difference that it can execute in parallel with its caller. Like a subroutine, a task has arguments passed by its caller, and may call other subroutines. A parallel application may originate a single task to execute the program that is equivalent to a sequential execution, or may originate multiple tasks for the computations. That originates multiple tasks to execute an application is called multitasking.

A task may have private variables, and it can access global data, sharable with other tasks. Task can create other tasks. Creating a task has some overhead, but initialization only takes place at startup and does not affect the performance of the task. Instructions of tasks are sharable, tasks are less costly to create, delete, and schedule. When user starts an application, operating system creates a master task to run the program. The master task may create other tasks. The tasks that were created by the master task attach to the master task and can, in turn, create other tasks. All the tasks are in the same level, there are no parent-child relationships among them. A task can terminate any task, including itself. A task with unfinished work cannot be terminated, but it can be terminated directly from the operating system.

A task is an environment for doing work in parallel. The life of a task goes through three important stages: creation, completion, and termination. A task must go through these three stages in order. Subroutines for task manipulations are as follows:

Dispatch_??
WaitForTask
WaitForAllTasks
TerminateTask
TerminateAllTasks
WaitForAndTerminateTask
WaitForAndTerminateAllTasks

1.3 Shared and Private Data

Sequential program does not distinguish if a variable is shared or private. However, a parallel (multitasking) program may execute multiple copies of a subroutine concurrently. In order to protect the content of all variables in each copy of the subroutine, each variable involved in the subroutine must be distinguished if it is shared or private. Shared data is accessible by all the tasks; while private data is accessible only by the task that allocates the private variable. Proper declaration of shared and private data is necessary in parallel programming.

By default, fortran variables are sharable (called static variables). Private data must be explicitly declared by the statement, for example, "AUTOMATIC", or subroutine declaring private variables should be with “recursive” attribute, for example “Recursive Subroutine" or "Recursive Function".

Only the shared variables are accessible by all tasks, which provide a way of communication between tasks. The shared data in a Fortran usually include the following types:

1. dummy arguments
2. variables defined in a common block
3. variables with initial values defined by the statement "DATA"
4. local variables declared with the attribute "SAVE"

Some Fortran compilers set the attribute "SAVE" as the default. Consult with Fortran manual for details.

It is a suggestion to apply name conventions for shared and private variables. For example, private variables may prefix with "Priv", or may end with the character $ as, "PrivA", "PrivB", "A$", "B$",,,,, and so on; while shared variables may prefix with "Shared". All the variables can be clearly identified if private or shared.

There are several advantages to sharing data:
1. It uses less memory than having multiple copies.
2. It avoids the overhead of making copies of data for each task.
3. It provides a simple and efficient mechanism for communication between tasks.

However, sharing data also has disadvantages: the most common one is a memory conflict. It may happen when one task is reading a memory while another task is writing on the same location simultaneously. This may lead to an unpredictable result, in order to avoid memory conflict synchronization is necessary.

In some situations, a private variable has more advantage than sharing data. For example, a loop variable should be a private data. If a loop variable is shared, a memory conflict must take place. Temporary variables are private. Distinguishing sharable and private data is very important in multitasking programming. A common error in parallel programs is that "*data that should not be shared is shared, and data that should be shared is not shared*."

1.4 Data Dependencies

Multitasking programs deal with shared as well as private data. Private data belongs to the task itself, which does not affect another tasks. Tasks may access shared data, and there is data dependency. For example, if task "a" accesses to elements X(1), X(2),,,, X(10), and Task "b" accesses elements X(8), X(9), X(10), X(11),,,, X(13), then Tasks "a" and "b" depend on data X(8), X(9), and X(10), data dependencies. For a dependent data for example X(8), if the data is being read by task “a” and written by task “b” simultaneously, this must lead to the wrong result. To ensure correct results, code sections that contain such dependencies cannot be executed simultaneously, but by one task at a time. Code sections containing dependencies are so-called critical sections.

The degree of data dependency is the ratio of the number of data dependent to the number of shared data. In the example, the task "a" has 30% data dependencies, and the degree of data dependencies in task "b" is 50%. Less data dependency may yield higher efficiency. Data dependence cannot be avoided in most scientific and engineering computing. Users can apply parallel lock to critical section to ensure the shared data can be accessed in order.

A critical section, for example, consists of the following steps:

1. to lock on parallel lock
2. to access shared data
3. to unlock parallel lock

Once a task has locked on the parallel lock, the task is the only one being able to execute the subsequently critical section. Another tasks cannot enter the critical section to access shared data until the locking task (or called lock holder) unlocks the parallel lock. The dependent data are allowed to be access only in a critical section, by which memory conflict is avoided. MTASK provides the following subroutines for parallel locks:

1. CreateParallelLock
2. LockonParallelLock
3. UnlockParallelLock
4. DeleteParallelLock

1.5 Task Synchronization

Task synchronization and data dependency are two common problems to block a task to execution. Data dependency limits dependent codes to be executed in critical section by one task at a time. Synchronization makes tasks wait for a permission to enter a code area for execution. This may happen, for example, a task is preparing for the needed data for the other tasks, and then all the other tasks have to wait for the availability of data.

Synchronization usually applies to the two following occasions:

1. A task needs the data computed in another task so as to wait for the completion of that task.
2. A task waits for the arrival of needed data.

A parallel event is a flag that handles synchronization in task, which counts on a cycle. A cycle of an event begins with an initialization, and then posts the event when a task has completed a predefined work, and finally ends with a request to wait for the completion of the cycle. A parallel event must create before use, and must be deleted when it is not needed any more. MTASK provides the following functions for manipulation of parallel events:

1. CreateParallelEvent
2. InitializeParallelEvent
3. PostParallelEvent
4. WaitForParallelEvent
5. IfCompleteParallelEvent
6. CompleteParallelEvent
7. DeleteParallelEvent

1.6 Function Partitioning

Some applications may contain several procedures that could run simultaneously. For example, a projectile contains

1. a procedure to calculate the velocity,
2. a procedure calculating the position.

The velocity and position of a projectile can be written in two independent equations, and they may calculate simultaneously. This example can be divided into two independent functional units for computing. This method is sometimes called heterogeneous multitasking, because it involves different tasks executed in parallel. Function partitioning is suitable for applications having independent functions.

1.7 Data Partitioning

Opposed to function partitioning, data partitioning is well suitable for applications that repeatedly calculate a large collection of data on certain subroutines, for example the addition of two vectors {C}={A}+{B}. We may develop a subroutine that receives partial elements of {A} and {B}, and then calculates the corresponding elements of {C}. Such kind of subroutine may create as multiple tasks in the same code, but with a different subset of {A} and {B}. This method is sometimes called homogeneous multitasking, because it involves tasks having an identical procedure executed in parallel.

Some applications may be programmed on both function partitioning and data partitioning; while others may be programmed on either function partitioning or data partitioning. Most parallel algorithms are based upon the technique of data partitioning. If both programming methods can be effective, the data partitioning fits more applications and offers the following advantages over function partitioning:

1. Minimal programming effort is required, because tasks having the same subroutines.
2. It is easy to balance loads among processors.
3. When system adding or removing processors, programs may automatically adapt to the number of processors.

1.8 Data Distribution

Subroutines in data partitioning may create multiple tasks. Each task has an individual set of data. The data can be distributed among tasks by two methods: static distribution and dynamic distribution.

In static distribution, the data distributed to a task are predetermined before the task creates. For example, if five tasks originate to execute a set of data, the data may be equally divided into five subsets; each task then has an individual subset. Static distribution cannot automatically balance workload among processors during execution. The reason is that processor is shared by multi-programs.

In dynamic distribution, the data are distributed when needed. There are no data distributed to tasks when the tasks create. Dynamic distribution produces dynamic load balancing, and keeps all tasks working as long as there is work to do. Static distribution produces static load balancing. Dynamic distribution entails more overheads than static distribution. Each time when a set of data is distributed to a task, the task must check the overall status of the data to make sure if there is work to do.

For developing a parallel application, programmer must determine which method offers the best load balancing with least overhead. If all the processors have balanced workload during execution, static distribution is more efficient. However, it is not always possible to keep workload balancing on static distribution when executing with another programs. If there is a large variation of workload between processors, dynamic distribution is more efficient. Even though dynamic distribution takes more overhead, dynamic distribution is more practical than static distribution on a multiprogramming system.

1.9 Types of Parallelism

A parallelism is a segment of statements executing a set of independent data. The efficiency of a parallel program relies on the parallelism and degree of data dependencies. As introduced previously, degree of data dependencies is an index that indicates the difficulty to parallel processing. This section introduces another parameter to control efficiency. A parallelism is characterized in terms of grain size, the number of operations of a parallelism, and can be divided into four types as follows:

1. Very coarse parallelism: Grain size is over 2000.
2. Coarse parallelism: Grain size is between 200 and 2000.
3. Medium parallelism: Grain size is between 20 and 200.
4. Fine parallelism: Grain size is less than 20.

A coarse parallelism means that a relative-low overhead is required to perform the parallelism; while a fine parallelism means a relative-high overhead is necessary. Each type of parallelism may have the best performance if the corresponding facilities are available. For example, distributed processing is most well suited for very coarse parallelism; vectorization may make fine parallelism work well. How to determine grain size? User should think if it is worth parallelizing 2 operations if overhead takes more than 2 operations. Parallel program may contain sequential segments that are limited to a task only. A task may have three types of statement segment:

1. parallelism that allows all tasks to concurrently execute,
2. critical section that allows all tasks to execute, but one at a time,
3. sequential segment that allows only one task to execute.

1.10 Lifetime of Variables

Variable has a certain lifetime that is the period when the variable exists. The variable lifetime is also an important subject that has to be considered in parallel programming. Variable lifetime can be classified into the following:

1. **Program-term lifetime**: Variables that have a program-term lifetime are allocated when the program begins and remains until the program ends. Each variable with a program-term lifetime has an identical address if more than one task executes concurrently. Program-term variable is sometimes called sharable variable, or global variable.
2. **Subroutine-term lifetime**: Variables, that have a subroutine-term lifetime, come into being when the subroutine begins, and disappear when the subroutine ends. A variable with a subroutine-term lifetime may have an individual address allocated at each start of the subroutine. If several copies of the subroutine execute concurrently, each copy of the subroutine has an individual address for the subroutine-term variables. This type of variable is declared with the statement of "automatic" in a Fortran program.

Life cycle of variables don't play significant role in sequential programs. However, in a parallel code, a lifetime type sometimes cannot be switched from one to the other. A variable with the wrong type of lifetime may lead to a fatal error in parallel processing. A private variable must have a subroutine-term lifetime. A shared data must have a program-term lifetime.

The lifetime of variables has to be carefully declared and dealt with in a parallel program, especially for subroutine-term variables. There is another type of error that may happen to a subroutine-term variable. This error is called a short-lived variable that disappears when another subroutines still need it. For example, if a subroutine "A" creates two tasks which execute subroutine "B". Subroutine "A" passes a subroutine-term variable "X" to subroutine "B". The variable "X" is allocated by subroutine "A". However, subroutine "A" ends before one or both copies of subroutine "B" complete their work. Under this circumstance, what will be happened? The end of subroutine "A" makes variable "X" disappears, but subroutine "B" still needs variable "X" to complete the work. Then, an error may occur due to the short life of variable "X".

A parallel program does not allow a short-lived variable to exist. There are three methods to prevent short-lived variables:

1. Try to dispatch all the subroutines in the main program. If that so, all the variables passed to the dispatched subroutine exist as long as the end of the application.
2. Declare the variables, which are passed to the dispatched subroutine, as program-term variables if possible, such that the variables may exist during the application execution.
3. Apply task synchronization to prevent the dispatcher from ending itself until all the copies of dispatched subroutine complete their work.

Chapter 2. Manipulation of Tasks

A task starts with a subroutine that may call other subroutines. All the parameters passed to the entry subroutine are sharable among tasks. A task is an environment to start executing a tasking subroutine in parallel. Tasking subroutines can be programmed in data partitioning or function partitioning. This chapter assumes that a tasking subroutine is ready, and discusses how to execute and synchronize tasks. Eight subroutines are introduced for the following purpose

1. creation of tasks
2. waiting for the completion of a task
3. termination of tasks

2.1 Creation of Tasks

This section introduces subroutine *Dispatch_??,* which creates a task. Once the tasking subroutine starts executing, the caller continues its execution. MTASK allows an application to have 64 tasks at a time. The syntax is as follows:

```
CALL Dispatch_??(TaskID, Procedure, Argu1, Argu2,,,,,  ArguN)
```

where

1. ?? is the number of arguments passed to the subroutine "Procedure". The number of arguments cannot be greater than 52. For example, if subroutine Procedure has 2 arguments, the calling statement should be written as

   ```
   CALL Dispatch_2(TaskID, Procedure, Argu1, Argu2)
   ```

   Here emphasizes again that ?? is the number of arguments to "Procedure", not to Dispatch_2. In the example, Dispatch_2 has 4 arguments. If "Procedure" does not have an argument, the syntax is as:

   ```
   CALL Dispatch_0(TaskID, Procedure)
   ```

2. If the request is successful, TaskID returns a 4-byte positive integer identifying the task. If the request fails that may occur to originate more than 64 tasks at a time, TaskID returns a zero.
3. "Procedure" is the name of subroutine to be executed in parallel. The subroutine must be declared by the statement "External" in a Fortran program as

   ```
   EXTERNAL Procedure
   ```

   Or, declared as a dummy procedure in Fortran 90.

   The subroutine "Procedure" can call other subroutines. All the subroutines directly or indirectly called by "Procedure" belong to the same task. Execution of a "RETURN" statement in the subroutine "Procedure" ends the task.

4. Argu1, Argu2,,,, and ArguN are actual arguments that are being passed to subroutine "Procedure", and are sharable among tasks. The arguments have some restrictions, and may not be any of the following:

   ....A loop index variable
   ....An expression requiring evaluation at run-time
   ....A constant
   ....A character string
   ....A short-lived variable

The following provides some examples of subroutine Dispatch_??. Each example assumes that variables have been properly declared.

Example A: Are the following statements correct?

```
DO K = 1, NumberOfTasks
   CALL Dispatch_3(TaskID(K) ,Procedure, A(K))
END DO
```

No, because subroutine Procedure has one argument, the statement should correct as "CALL Dispatch_1(...."

Example B: Are the following statements OK?

```
DO K = 1, NumberOfTasks
   CALL Dispatch_2(TaskID(K), Procedure, A(K), K)
END DO
```

It is not OK. The second argument "K" that is passed to the subroutine "Procedure" is the loop index. Can we modify the previous loop into the following?

```
DO K = 1, NumberOfTasks
   J=K
   CALL Dispatch_2(TaskID(K), Procedure, A(K), J)
END DO
```

where "J" is not the loop index. It does not make the right modification, because the variable "J" is shared among tasks, and varies with the loop index. "J" is uncertain in the subroutine "Procedure". This is one of common problems that beginners may encounter.

A correct modification should introduce an array, for example "JTEMP", and then assign the loop index onto an element of "JTEMP", i.e. JTEMP(K)=K, which may be written as:

```
DO K = 1, NumberOfTasks
   JTEMP(K)=K
   CALL Dispatch_2(TaskID(K), Procedure, A(K), JTEMP(K))
END DO
```

Example C: Is the following statement correct?

```
CALL Dispatch_1(TaskID(K), Procedure, 3.0+A(K))
```

No, because the argument 3.0+A(K) should be evaluated at run-time.

Example D: Is the following statement correct?

```
     CALL Dispatch_1(TaskID(K), Procedure, 5)
```

No, because the argument 5 is a constant.

Example E: Is the following statement correct?

```
     CALL Dispatch_1(TaskID(K), Procedure, 'Input.dat')
```

No, because the argument 'Input.dat' is a character string. Dispatch_?? does not pass a character string. If it is necessary to pass a character string, define a one-byte integer array, and then assign the decimal value of character string onto the 1-byte integer array, and pass the 1-byte integer array to subroutine "Procedure". The 1-byte integer array is reversed into a character string in subroutine "Procedure".

Example F: Are the following statements correct?

```
     Recursive SUBROUTINE AAA(Argu_1,,,,,
     INTEGER (4) :: TaskID(,,,,
     Integer (4) :: M
     EXTERNAL TEST
     :
     :
     DO K = 1, NumberOfTasks
        CALL Dispatch_1(TaskID(K), TEST, M)
     END DO
     RETURN
     END SUBROUTINE AAA
```

No. Once the subroutine "AAA" completes dispatching subroutine "TEST" in the loop, subroutine "AAA" executes the statement of "RETURN", which destroys variable "M" that is a subroutine-term lifetime. However, subroutine "TEST" still needs the variable "M" to complete the work. There are two ways to correct this problem. The first one is to declare the variable "M" with the "SAVE" attribute. That makes variable "M" a program-term variable.

The second method is to apply a task synchronization that prevents the dispatcher from executing the "RETURN" statement until all the tasks complete their work. That may be written as:

```
     Recursive SUBROUTINE  AAA(Argu_1,,,,,
     INTEGER (4) :: TaskID(,,,,,
     Integer (4) :: M
     EXTERNAL TEST
     :
     :
     DO K = 1, NumberOfTasks
        CALL Dispatch_1(TaskID(K), TEST, M)
     END DO
     DO K = 1, NumberOfTasks
        CALL WaitForAndTerminateTask( TaskID(K) )
```

```
      END DO
      RETURN
      END SUBROUTINE AAA
```

The subroutine "WaitForAndTerminateTask" blocks the caller until the specified task completes its work.

Example G: What is the difference between the following two segments?

```
      [segment 1]
         DO K = 1, NumberOfTasks
            CALL Dispatch_??(TaskID(K), SubA,,,,)
         END DO
         CALL WaitForAndTerminateAllTasks

      [segment 2]
         DO K = 1, NumberOfTasks-1
            CALL Dispatch_??(TaskID(K),SubA,,,,)
         END DO
         CALL SubA(,,,)
         CALL WaitForAndTerminateAllTasks
```

Both of them execute "NumberOfTasks" copies of subroutine "SubA" concurrently. The difference is on master task. In the first segment, the master task does not execute subroutine "SubA". The master task in "Fragment 2" executes subroutine "SubA". "Segment 2" takes less overhead and is more efficient because "Segment 1" originates one more task.

2.2 Waiting For the Completion of Tasks

The "WaitFor" subroutines provide a way to allow a task to wait for the completion of work assigned to tasks. These subroutines may check if a task has completed its work. There are no parent-child relationships among tasks. Any task may issue a request to wait for another tasks. However, don't let a task wait for the completion of itself. Waiting for itself will result in a deadlock. A deadlock causes a program not to execute. An example is as follows: task "X" issues a request to wait for the completion of task "Y", while task "Y" is waiting for task "X". Apparently, neither task "X" nor task "Y" may complete. Here introduces three subroutines for the WaitFor request:

1. WaitForTask
2. WaitForAllTasks.

The syntax of WaitForTask is as follows:

CALL WaitForTask(TaskID)

where TaskID is the identifier of the task that must complete its work to satisfy the statement. If the task identified by TaskID has been complete, the caller returns from the subroutine immediately; otherwise the caller is blocked until the work assigned to the task identified by TaskID has been completed. The request to WaitForTask is ignored, if the TaskID is invalid. This function makes sure if a certain task has completed its work.

The syntax of routine WaitForAllTasks is as follows:

```
CALL WaitForAllTasks
```

There are no arguments required in the routine. The caller is blocked until all the tasks that are created by Dispatch_?? have finished their work. A task created by Dispatch_?? cannot issue a request to WaitForAllTasks. That may make the caller wait for the completion of itself, and results in a deadlock. The WaitForAllTasks subroutine must be called from the master task. A deadlock sometimes may be solved by means of:

1. Try to call WaitFor in the master task.
2. If it cannot help but request a WaitFor in a task created by subroutine Dispatch_??, be sure not to wait for itself.

Multiple WaitFor requests to the same task is ignored; for the following example,

```
CALL WaitForTask(1)
CALL WaitForTask(1)
CALL WaitForTask(1)
```

Only the first WaitFor request is actually performed. The second and third requests are ignored. The WaitFor subroutines clear the status for termination. Once a task has completed its work, the task can be terminated. WaitFor and Terminate work together as a pair.

2.3 Termination of Tasks

An attempt to delete a task with unfinished work is ignored. There are two types of subroutine for terminating tasks, TerminateTask and TerminateAllTasks.

The syntax for TerminateTask is as follow:

```
CALL TerminateTask(TaskID)
```

This function deletes the task identified by TaskID. If TaskID is a valid identifier, the caller will be blocked until the task identified by TaskID is deleted. If TaskID is an invalid identifier, i.e., it is not a task created by Dispatch_??, the calling request is ignored. This function usually follows WaitForTask.

The syntax for TerminateAllTasks is as follow:

```
CALL TerminateAllTasks
```

This routine does not require an argument, and deletes all the tasks created by Dispatch_??. This subroutine usually follows the WaitForAllTasks subroutine in the master task as:

```
CALL WaitForAllTasks
CALL TerminateAllTasks
```

The most common error resulted from Terminating subroutines is an attempt to delete a task with unfinished work.

2.4 Waiting For and Terminating Tasks

As mentioned before, the WaitFor and Terminate subroutines work together as a pair, i.e.,

```
CALL WaitForAllTasks
CALL TerminateAllTasks
```

MTASK has subroutines those combine two functions, for example, WaitForAndTerminateTask and WaitForAndTerminateAllTasks.

The syntax for WaitForAndTerminateTask is as follow:

```
CALL WaitForAndTerminateTask( TaskID )
```

where TaskID is the identifier of the task. This function waits for the completion of work assigned to the task identified by TaskID, and then terminates the task. This function is equivalent to the following statements:

```
CALL WaitForTask( TaskID )
CALL TerminateTask( TaskID )
```

The syntax for WaitForAndTerminateAllTasks is as follow

```
CALL WaitForAndTerminateAllTasks
```

which is equivalent to the following statements:

```
CALL WaitForAllTasks
CALL TerminateAllTasks
```

Chapter 3. Manipulation of Parallel Locks

Parallel locks handle critical sections. A shared-data is not allowed to access by multiple tasks simultaneously. This chapter introduces subroutines for manipulation of parallel locks. When a task enters the critical section, the task must lock on the parallel lock. And, when the task exits the critical section, the task unlocks the lock. Four subroutines are introduced as follows:

1. CreateParallelLock
2. DeleteParallelLock
3. LockonParallelLock
4. UnlockParallelLock

3.1 Definitions

The syntax for CreateParallelLock is as follows

```
CALL CreateParallelLock(LockID)
```

This function creates a parallel lock, and returns a 4-byte positive integer LockID as the identifier of the lock. Mtask allows an application to create 64 parallel locks simultaneously. The lock identifier requires when obtaining, releasing, or termination of the lock. A lock has to be created by this function before use. This function is usually called from the master task.

The syntax for DeleteParallelLock is as follows:

```
CALL DeleteParallelLock(LockID)
```

This function deletes the parallel lock identified by LockID. A parallel lock must be deleted when it is no longer needed in an application so as to free memory. This function is usually called from the master task.

The syntax for LockonParallelLock is as follows:

```
CALL LockonParallelLock(LockID)
```

This function is called when the caller attempts to enter the critical section. If there is no other task holding the lock, the caller may immediately execute the critical section and becomes a lock holder; otherwise, the caller is blocked until the current lock holder releases the lock. A call to the subroutine LockonParallelLock" is the first statement of a critical section. This function is usually called in a tasking subroutine.

The syntax for UnlockParallelLock is as follows:

```
CALL UnlockParallelLock(LockID)
```

This function releases the lock identified by LockID. Only the current holder may release the parallel lock. Once the lock has been released, one of the other tasks if they are waiting for the lock may obtain the lock and becomes a new holder. A call to routine "UnlockParallelLock" means an end of a critical section. A pair of "LockonParallelLock" and "UnlockParallelLock" control access of a shared data area. That ensures that the shared data is accessed by one task at a time. This function is usually called in a tasking subroutine.

3.2 Critical Section

Critical section has example statements as:

```
CALL LockonParallelLock(LockSectionA)
   [critical operations, i.e.,
               access a shared read-write data area]
CALL UnlockParallelLock(LockSectionA)
```

It begins with a call to the subroutine "LockonParallelLock", and ends at a call to the subroutine "UnlockParallelLock". When a task returns from the subroutine LockonParallelLock, it is the only task (called lock holder) that executes the critical section. Before the lock released by the lock holder, all the other tasks that attempt to enter the critical section are blocked in the subroutine LockonParallelLock until the holder releases the parallel lock in subroutine UnlockParallelLock. And, one of the waiting tasks becomes a new lock holder that immediately executes the critical section. The new holder keeps locking out the other waiting tasks off the critical section. This mechanism protects the critical section to be executed by one task at a time.

If a lock holder never released its lock, a deadlock may occur. That inappropriately handles parallel locks may result in a deadlock. The operations in critical section have to be reduced to the minimal so as to improve the performance.

Lock contention is also an important factor that needs to be dealt with. In the situation of low lock contention, the caller won't be blocked when requesting a lock. There are two methods to reduce lock contention:

1. Try to reduce the operations in a critical section to the minimal.
2. Try to increase the grain size of a parallelism, such that the time spent in a critical section becomes relatively small.

Chapter 4. Manipulation of Parallel Events

This chapter introduces subroutines for task synchronization. Parallel events handle task synchronization. MTASK has nine subroutines for parallel event, which are as:

1. CreateParallelEvent
2. InitializeParallelEvent
3. PostParallelEvent
4. WaitForParallelEvent
5. IfCompleteParallelEvent
6. CompleteParallelEvent
7. DeleteParallelEvent

4.1 A Cycle of an Event

A parallel event counts on cycle. A cycle means all the tasks have satisfied a request. When completing a predefined work, a task must post the parallel event once. MTASK accumulates the posts. Once the event's post-count reaches the specified counts, a cycle of parallel event is satisfied. A cycle of an event goes through the following steps:

1. an initialization that sets a requirement for tasks to reach
2. post of event when a task satisfies a count
3. the end of the event that waits for the completion of the cycle

For example, if there are ten subsets of data to be calculated. A cycle is initialized to ten. When finishing a subset, a task posts the parallel event once. When the post-count is less than ten, tasks cannot leave the parallelism, but keep working and waits until the post-count reaches ten. This mechanism allows tasks to wait for the completion of a parallelism. When the post-count reaches ten, a cycle of the parallel event is complete. This indicates that the ten subsets are complete, and all the tasks are free for the next parallelism. When applying parallel event to synchronize tasks, the count can be defined to several ways.

4.2 Definitions

The syntax for CreateParallelEvent is as follow:

```
CALL CreateParallelEvent(EventID)
```

This function creates a parallel event. MTASK allows an application to have 64 parallel events at a time. If calling request is successful, EventID returns a 4-byte positive integer identifying the event. If calling request fails, the variable EventID returns a zero. The identifier is required when manipulating the event. A parallel event must be created by this subroutine before use. This function is usually called from the master task.

The syntax for InitializeParallelEvent is as follow:

```
CALL InitializeParallelEvent(EventID,Count)
```

This function initializes a cycle of parallel event. EventID is the identifier of the parallel event. Count is a 4-byte positive integer. When completing a cycle, an event can be re-initialized into another cycle. If an event has not completed a cycle, re-initializing the event may lead to an unpredictable result.

The syntax for PostParallelEvent is as follow:

CALL PostParallelEvent(EventID)

This subroutine posts the specified event identified by EventID.

The syntax for WaitForParallelEvent is as follow:

CALL WaitForParallelEvent(EventID)

This function makes a task synchronize with the other tasks. EventID is the identifier of the parallel event. If a cycle of the event has completed, caller immediately returns; otherwise, the caller will be blocked until the cycle is complete. The completion of a cycle means the specified event's post-count is reached the "Count".

The syntax for IfCompleteParallelEvent is as follow:

CALL IfCompleteParallelEvent(EventID,Yes)

This function extends a cycle of the event to execute a sequential segment that follows a parallelism. A tasking subroutine may contain three kinds of segment:

1. parallelism that allows all tasks to execute concurrently
2. critical section that allows all tasks to execute but one at a time
3. sequential segment that allows only one task to execute

Synchronization can be handled by parallel event. A critical section is controlled by parallel lock. Subroutine "IfCompleteParallelEvent" is applied to the combination of parallelism and sequential segment. If a sequential segment follows a parallelism, "IfCompleteParallelEvent" provides a mechanism for task to execute the sequential segment as:

1. to wait for the completion of a cycle
2. to return a flag "Yes" that directs a task to execute the sequential segment or just to jump over the sequential segment.

"Yes" is a 4-byte logical variable. Calling this routine, only one task will get a false "Yes". The task that returns a false "Yes" must continue executing the sequential segment, and the tasks that return a true "Yes" must jump over the sequential segment. Until one task completes the sequential segment, the other tasks cannot be free. For example,

```
:
:
CALL PostParallelEvent(EventID)
CALL IfCompleteParallelEvent(EventID,Yes)
IF( .NOT.YES ) THEN
```

```
:
:  [sequential segment]
:
   CALL CompleteParallelEvent(EventID)
END  IF
:
:
```

When the event's post-count reaches the "count", only one task executes the sequential segment. The subroutine "CompleteParallelEvent" must be the last statement in the block of "IF(.NOT.YES)", that frees the other tasks blocked in routine "IfCompleteParallelEvent".

The syntax for CompleteParallelEvent is as follow:

CALL CompleteParallelEvent(EventID)

EventID is the identifier of the parallel event. This function frees the tasks blocked in the subroutine "IfCompleteParallelEvent".

The syntax for DeleteParallelEvent is as follow:

CALL DeleteParallelEvent(EventID)

This function deletes the specified event identified by EventID, and is usually called by the master task. This subroutine is called if an application does not need the event any more. An event is deleted once.