Multithreaded Programming in XL Fortran Version 5

By Julian Wang

To facilitate explicit parallelization of Fortran programs, XL Fortran Version 5 provides a Fortran 90 module to interface to the pthread *library. The module provides procedures that handle the details of Fortran-C interlanguage calls, making it easy for Fortran codes to exploit and use threads.*

Parallel programming is essential to exploit the power of multiprocessor systems. XL Fortran Version 5 supports two fundamental approaches for parallel programming. First, the programmer can insert appropriate directives into the source code to guide the compiler in parallelizing nested loops. Second, the programmer can use the thread library to explicitly parallelize the computations. The first approach corresponds to data parallel programming, while the second approach corresponds to task parallel programming models.

The choice between the two approaches largely depends on the nature of the application, more specifically, the computation model used in the underlying algorithm. It may be beneficial to employ task-level parallelism to map large chunks of the application and use data parallelism to map iterations of nested loops within the chunks. Thus, the two approaches are not mutually exclusive, and in fact, XL Fortran V5 allows the programmer to mix the two approaches in the same application.

The XL Fortran V5 compiler is effective in parallelizing applications by automatically detecting data parallelism. However, it may be necessary to explicitly parallelize programs using task parallelism for improved performance. Although it is tedious to inform the compiler about task-level parallelism, often it is the only approach to exploit parallelism in certain applications.

The facilities in the XL Fortran V5 compiler for explicitly specifying parallelization provide the programmer finer control over overlapping operations so as to obtain the best performance possible. The `pthread` library on AIX provides necessary and portable facilities to parallelize a program. However, for native Fortran 90 programmers, a Fortran 90 interface to the `pthread` library is highly desirable for ease of programming. This article summarizes the `pthread` library facilities in general, and introduces a Fortran 90 interface module, `f_pthread`, provided with the XL Fortran V5 compiler.

AIX pthread Library

The `pthread` library on AIX is based on the emerging POSIX™ 1003.1c industry standard. It provides an object-oriented interface to the application. The programmer manipulates opaque objects through object-type related operations. The library defines both the object types and the allowed operations. The `pthread` library has three pairs of object types:

- Threads and thread attributes objects
- Mutexes and mutex attributes objects
- Condition variables and condition attributes objects

Julian Wang

Creating an object requires the creation of an attributes object. The attributes object specifies certain aspects of the characteristics of the created object. See *AIX Version 4.1 General Programming Concepts: Writing and Debugging Programs* for more information on the characteristics that can be specified by the programmer for each type of object.

Typically, an attributes object is created with attributes having default values. Individual attributes in the attributes object can then be modified using operations provided by the library. Once the objects are created, they are not affected by destroying or modifying the attributes object used to create them.

Thread Creation and Termination

A *thread* is an independent execution sequence in a process. The `pthread` interface allows more than one thread to be active at a time. Executing multiple threads concurrently can possibly reduce the time to complete an application on a multiprocessor system. Multiple threads in a process share the system resources possessed by the process, including the memory.

A thread is created by a call to `pthread_create`, where one of the arguments is the routine that becomes the entry point for the created thread. The thread is terminated automatically when the thread returns from this routine. In addition, a thread can explicitly terminate itself by calling the `pthread_exit` procedure. This allows resources associated with the thread to be available in a timely manner.

A thread can also be terminated by any other thread in the process. Because all threads share the same data space, it is important to perform cleanup operations at the termination time. For this purpose, the `pthread` library provides a procedure to register cleanup handlers for a thread.

It should be noted that the initial thread is created by the system when the process starts. If this initial thread terminates by returning from the main program, then the process itself is terminated, including all its threads. If the initial thread terminates by calling `pthread_exit`, it will not affect other threads in the process.

Thread Synchronization

When multiple threads are active simultaneously, the ability to synchronize their activities is necessary. However, data inconsistencies may result in these interactions because of race conditions. A race condition occurs when two or more threads need to perform operations on the same set of data items, but the results of computations depend on the order in which these operations are performed.

Another form of interaction among threads is through explicit communication about the events that occur. This requires threads to be able to wait for other threads to complete an activity, including a thread's termination. When the condition occurs, then the waiting threads should be informed of the condition.

The `pthread` library provides a primitive synchronization mechanism for each of the purposes above, mutexes and condition variables, respectively. There are procedures to create, manipulate, and destroy mutex or condition variable objects. Mutex objects can be created by calling `pthread_mutex_init`. The mutex objects can then be locked by calling `pthread_mutex_lock` or `pthread_mutex_trylock` to prevent data inconsistencies from happening. The mutex locks can be released by calling `pthread_mutex_unlock`. Finally, mutexes can be deleted by the procedure `pthread_mutex_destroy`.

Threads can call `pthread_cond_wait` or `pthread_cond_timedwait` to wait on a condition variable after it is created by `pthread_cond_init`. When the condition is satisfied, a thread can use `pthread_cond_signal` or `pthread_cond_broadcast` to inform the waiting thread(s) of the event. After using a condition variable, its associated resources can be reclaimed by calling `pthread_cond_destroy`.

Thread Scheduling

Threads require resources such as processor time, memory, and file descriptors in order to execute properly. The manner in which the threads are scheduled to use those

resources can have a profound effect on the performance of programs. The pthread library provides several facilities to handle and control the scheduling of threads, including:

- Setting scheduling attributes when creating a thread
- Dynamically changing the scheduling attributes of a created thread
- Defining the effect of a mutex on the thread's scheduling when creating a mutex
- Dynamically changing the scheduling of a thread during synchronization operations

The last two types of controls are known as *synchronization scheduling*. Synchronization scheduling is not yet available on AIX, so it will not be discussed further.

Three scheduling parameters are associated with each thread: contention scope, scheduling policy, and scheduling priority. Usually, the library provides default values for these parameters, which are sufficient for most cases. However, the scheduling parameters can also be set either by using the thread attributes object before the thread's creation, or by dynamically changing attribute values during the thread's execution. The procedure pthread_attr_getschedparam can be called to check the current setting of scheduling attributes in a thread attributes object; whereas pthread_attr_setschedparam can be used to change those attributes' values. After a thread is started, pthread_getschedparam can be used to retrieve the current scheduling attributes of the thread, and pthread_setschedparam can be called to change them.

Advanced Features

The thread library provides some advanced features to be used by more experienced programmers. These include the one-time initialization facility, thread-specific data support, and thread stack management.

Some program libraries are designed for dynamic initialization; that is, the library will be initialized the first time a procedure in the library is called rather than requiring the caller to explicitly call an initialization procedure before any others. In multithreaded programs, however, this requires extra care since more than one thread can simultaneously call into the library. If not protected properly, the library may be initialized more than once. To make this task easier, the thread library provides a procedure called pthread_once. It will do nothing if other threads have called it already. This will make porting such a library to a multithreaded environment straightforward.

Threads require resources such as processor time, memory, and file descriptors in order to execute properly.

Many applications require that certain data be maintained on a per-thread basis across procedure calls. The thread-specific data interface is provided to meet these demands. Essentially, thread-specific data can be viewed as a two-dimensional array, with keys serving as the row index and thread IDs as the column index. Before manipulating thread-specific data, threads must create (or know) the thread-specific data keys designating the data by calling pthread_key_create. It should be noted that the keys are common to all threads in a process, only the data associated with them are private to each thread. Thread-specific data can be manipulated by the pthread_setspecific and pthread_getspecific procedures, with a key as one of the arguments.

Thread stacks are usually automatically and transparently managed by the system. In general, these stacks are sufficient for most applications. However, they may be too restrictive for certain applications. Using advanced thread attributes, it is possible for the user to control the size of the stack (for example, pthread_attr_setstacksize).

The f_pthread Fortran 90 Module

As it is, the pthread library on AIX is easy to use with the C programming language (or its derivatives). But it is not that straightforward to use within Fortran applications. At least three obvious issues need to be addressed to make it friendly for Fortran programmers:

- Provide derived data types for each object type mentioned above
- Observe possible differences in argument-passing conventions between Fortran and C
- Support native Fortran 90 data types and data structures (for example, the CHARACTER type and the array section construct)

A Fortran 90 module can be used to help resolve these issues. A module provides a means for packaging data types and procedures that operate on those types. This is why the Fortran 90 module f_pthread is provided in the XL Fortran Version 5.1 product. Before discussing the details of how f_pthread addresses the three issues, some general features of the module should be noted.

There is a one-to-one correspondence between the entities provided by the f_pthread module and those in the pthread library. Entities are named by prefixing f_ to the corresponding pthread name (except for constants). For example, there is a Fortran 90-derived type f_pthread_attr_t to be used in Fortran programs as pthread_attr_t is used in C programs. f_pthread_create can be called to create a thread using the module instead of pthread_create from the library. The module procedure will return whatever value is returned from the library routine. The error code constants are also provided in this module. EINVAL from this module has the same value as is defined in the system header file. These symbolic names will make Fortran programs more portable.

Data Types

The object-oriented interface provided by the pthread library is intended to make programs that use this interface portable across different platforms. It can also shield the application programmers from the exact definitions of the object types on each platform. As a matter of fact, the definitions of the object types are highly likely to be different on different systems, and may change from release to release on a particular one. This poses an obstacle to native Fortran programmers trying to do multithreaded programming without basic support, such as an f_pthread module, from the language product. The C struct definitions must be mapped very carefully into Fortran 90-derived types, observing padding and alignment rules. This is non-portable and tedious, to say the least.

With the f_pthread module, it is easy to use pthread object types in Fortran, and the portability across AIX® levels is guaranteed by the product. The example in Figure 1 shows how to declare a thread attributes object, a thread object, and a mutex object.

```
use f_pthread
type(f_pthread_attr_t)  obj_attr
type(f_pthread_t)       obj_thread
type(f_pthread_mutex_t) obj_mutex
```

Figure 1. Declaring thread attributes, thread, and mutex objects

Argument Passing

The differences of argument-passing conventions among programming languages makes interlanguage programming error-prone. It is possible to directly program in Fortran to the pthread library interface, but it is important to observe the argument-passing mechanisms from Fortran to C and from C back to Fortran. For example, in XL Fortran 5.1, the default is call-by-reference under most situations, whereas the default in C is call-by-value.

Although there are built-in functions that can be called to enforce calling convention consistency when language boundaries are crossed, programmers are

```
use f_pthread
type(f_pthread_attr_t)  obj_attr
integer old_state, new_state
integer any_err

! Initialize a thread attributes object
any_err = f_pthread_attr_init(obj_attr)

! Retrieve the default state
any_err = f_pthread_attr_getdetachstate(obj_attr, old_state)

! Set a new state: can be done either simply,
! any_err = f_pthread_attr_setdetachstate(obj_attr,  &
!       PTHREAD_CREATE_UNDETACHED)
! or,
new_state = PTHREAD_CREATE_UNDETACHED
any_err = f_pthread_attr_setdetachstate(obj_attr, new_state)
```

Figure 2. The detach state attribute in a thread attributes object

responsible for determining when they should be used. By using a module such as `f_pthread`, Fortran programmers are relieved of this problem since the module procedures provided worry about the interlanguage calling issues. Furthermore, the compiler is able to detect when incorrect arguments are used.

The code segment in Figure 2 shows how to manipulate the detach state attribute in a thread attributes object (error checking is not shown).

Native Fortran Constructs

Certain data types or data constructs are specific to Fortran and cannot be easily mapped to constructs in other languages. For example, the Fortran `CHARACTER` type has a length attribute, and Fortran 90 array sections have a shape attribute. The representation of these attributes is highly implementation dependent. It is a daunting job even for experienced programmers to figure out the exact details in a particular

```
use f_pthread
interface
    subroutine char_sub(charg)
        character(*) charg
    end subroutine
end interface
type(f_pthread_attr_t) attr
type(f_pthread_t) thread1
character(20), save:: c_arg
integer any_err

any_err = f_pthread_attr_init(attr)
any_err = f_pthread_create(thread1, attr, FLAG_CHARACTER, &
          char_sub, c_arg)
```

Figure 3. Subroutine requiring a CHARACTER argument

```
module global
    use f_pthread

    ! This program can be used to compute:
    !           c = a X b, where
    ! c is a matrix of dimension (x_size,z_size)
    ! a is a matrix of dimension (x_size,y_size)
    ! b is a matrix of dimension (y_size,z_size).
    ! The computation will be distributed among nthreads to be
    ! completed.
    integer nthreads, x_size, y_size, z_size
    parameter (nthreads=4, x_size=16, y_size=16, z_size=16)

    ! Set up a barrier for synchronization
    integer finish/0/
    type(f_pthread_mutex_t) ba_mutex/PTHREAD_MUTEX_INITIALIZER/
    type(f_pthread_cond_t)  ba_cond/PTHREAD_COND_INITIALIZER/
    type(f_pthread_t) threads(nthreads)

    ! Matrix mult_a and mult_c are arranged to take into account the
    ! fact that Fortran arrays are column-major. Spatial locality
    ! has substantial impact on the performance in this application.
    real(8) mult_a(y_size, x_size), mult_b(y_size, z_size)
    real(8) mult_c(z_size, x_size)
end module global

program example
    use global

    ! An explicit interface is required for an assumed-shape array
    ! argument.
    interface
        subroutine mult(rows)
            real(8) rows(:, :)
        end subroutine
    end interface
    integer i, ret

    ! Input mult_a and mult_b: not shown here.

    threads(1) = f_pthread_self()
    do i=2, nthreads
        ! We can do without a thread attributes object; the system
        ! default will be used. The workload distribution is cyclic.
        ret = f_pthread_create(threads(i), flag=FLAG_ASSUMED_SHAPE, &
                               ent=mult, &
                               arg=mult_a(1:y_size, i:x_size:nthreads))
    end do

    call mult(mult_a(1:y_size, 1:x_size:nthreads))

    ! Output mult_c: not shown here.
end program example

subroutine mult(rows)
    use global
    real(8) rows(:,:), part_r
    integer i, j, k, ret, which_thr, stride
    type(f_pthread_t) myself
```

Figure 4. Multithreaded Fortran 90 program (continued on following page)

```
    ! Find out who I am
    myself = f_pthread_self()
    do i = 1, nthreads
        if (f_pthread_equal(myself, threads(i))) then
            which_thr = i
        end if
    end do

    ! Do my part of work
    stride = 0
    do i = 1, ubound(rows, 2)
        do j = 1, z_size
            part_r = 0.0
            do k = 1, y_size
                part_r = part_r + rows(k, i) * mult_b(k, j)
            end do
            mult_c(j, which_thr+stride) = part_r
        end do
        stride = stride + nthreads
    end do

    ! Synchronize: main thread will return, but others will exit.
    ret = f_pthread_mutex_lock(ba_mutex)
    finish = finish + 1
    if (which_thr .eq. 1) then
        if (finish .ne. nthreads) then
            ret = f_pthread_cond_wait(ba_cond, ba_mutex)
        end if
    else
        if (finish .eq. nthreads) then
            ret = f_pthread_cond_signal(ba_cond)
        end if
    end if
    ret = f_pthread_mutex_unlock(ba_mutex)
end subroutine mult
```

Figure 4. Multithreaded Fortran 90 program (continued from previous page)

compiler. It is even more difficult to use them in calls to an interlanguage library such as the `pthread` library. Support is built into the `f_pthread` module, which allows a Fortran program to make use of these Fortran-specific features in calls to the `pthread` library.

Figure 3 shows how to create a thread, whose entry is a subroutine that requires a `CHARACTER` argument.

Application Examples

Figure 4 shows a multithreaded Fortran 90 program for matrix multiplication using the `f_pthread` module. The algorithm was chosen to show how native Fortran data constructs are supported and is not intended to be the best algorithm in terms of performance. For simplicity, all error checking is ignored. The comments in the source code in Figure 4 are helpful in understanding the algorithm.

Julian Wang, IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1V7. Mr. Wang is a staff software developer, working on the Fortran compiler and runtime. He has a BS in Computer Science from the University of Science and Technology of China, an MS in Computer Engineering from Academia Sinica, and an MS in Computer Science from the University of Toronto.